Spezifikation von Variabilität in automatisch bewerteten Programmieraufgaben

Robert Garmann[1]

HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS

Fakultät IV
Wirtschaft und
Informatik

Bericht

25. Januar 2018

Hochschule Hannover
Fakultät IV – Wirtschaft und Informatik
Ricklinger Stadtweg 120
30459 Hannover

[1] E-Mail: robert.garmann@hs-hannover.de

Zusammenfassung

Wir beschreiben eine Möglichkeit, Variationspunkte und deren Varianten in automatisiert bewerteten Programmieraufgaben zu spezifizieren. Solche Variationspunkte kommen bei individualisierbaren Programmieraufgaben zum Einsatz, bei denen jede Studentin und jeder Student eine eigene Variante einer Programmieraufgabe erhält. Die Varianten werden automatisch gebildet, indem an definierten Variationspunkten immer wieder andere, konkrete Werte eingesetzt werden. Schon bei sehr einfachen Aufgaben bestehen Abhängigkeiten zwischen den einzelnen Variationspunkten, die bei der Wahl der konkreten Werte zu berücksichtigen sind. Zudem kann die Menge der gültigen Werte auch bei einfachen Aufgaben so groß werden, dass die vollständige Auflistung aller Wertkombinationen an Ressourcengrenzen scheitert. Die vorgestellte Spezifikation verwendet eine kompakte und für Aufgabenautoren verständliche Sprache, die eine automatische Auswahl von korrekten, den Abhängigkeiten gehorchenden Wertekombinationen ermöglicht. Die Sprache ist unabhängig von den Erfordernissen eines bestimmten Autobewerters und versetzt Frontend- und Backendsysteme in verschiedenen technischen Ökosystemen in die Lage, ausgewählte Werte einer sehr großen Wertemenge zu generieren, deren Abhängigkeiten zu prüfen, sowie ggf. bestimmte Wertbelegungen in einem benutzerfreundlichen Dialog auszuwählen. Wir unterstützen Variationspunkte mit endlichen Mengen vorzugebender diskreter Werte sowie kontinuierliche Wertebereiche, die durch eine vorzugebende Anzahl von Samples diskretisiert werden. Wir beschäftigen uns insbesondere mit der Frage, wie lange Auflistungen gültiger Wertkombinationen durch die Angabe von Ableitungsvorschriften ersetzt werden können. Ein besonderes Augenmerk legen wir auf eine redundanzfreie Beschreibung der Variantenmenge. Die Notation setzt auf XML und Javascript in der Annahme, dass diese Technologien in allen beteiligten Systemen zur Verfügung stehen können.

Schlagwörter

Individuelle Programmieraufgaben, Grader, Autobewerter, E-Assessment, Variabilität

DDC Klassifikation

004 Datenverarbeitung; Informatik

GND-Schlagwörter

Programmierung, E-Learning, Computerunterstütztes Lernen, Übung <Hochschule>, Lernaufgabe

ACM CCS (2012)

• **Social and professional topics~Computer science education** • **Social and professional topics~Student assessment** • *Applied computing~Computer-assisted instruction* • *Applied computing~E-learning* • *Software and its engineering~Software product lines*

Inhalt

1 Einleitung

1.1 Individuelle, automatisiert bewertete Programmieraufgaben

Wir betrachten ein häufig anzutreffendes formatives Prüfungsszenario, in dem Informatik-Studierende kontinuierlich während des Semesters Programmieraufgaben zur Übung verpflichtend im Selbststudium bearbeiten. Die Übungsaufgaben haben die Erstellung (Synthese) eines Programms oder Programmfragments zum Gegenstand. Einfachere Aufgaben, die sich mit der Analyse eines gegebenen Quelltextes befassen, sind für das hier betrachtete unüberwachte Selbststudium nur bedingt geeignet.

Angesichts einer stark heterogenen Studierendenschaft ist es sinnvoll, Programmieraufgaben zu individualisieren. Bspw. kann es lernförderlich sein, individuelle Varianten einer Aufgabe so an Studierende auszugeben, dass der Schwierigkeitsgrad der Aufgabenvariante zum jeweiligen Lernstand des Lernenden passt. Auch könnten Studierende davon profitieren, mehrere, etwa gleich schwere Varianten ein und derselben Aufgabe nacheinander zu lösen, um ein zu lernendes Konzept intensiv zu üben. Nicht zuletzt erliegen erfahrungsgemäß etliche Studierende der Versuchung, fremde Lösungen als ihre eigenen einzureichen, wenn alle Studierenden die gleichen Aufgaben lösen müssen. Individuelle Aufgaben könnten dieses wenig lernförderliche Studierendenverhalten eindämmen helfen. Individualisierbare Aufgaben unterliegen zudem nicht dem Problem, dass sie durch Lösungsweitergabe von früheren an spätere Studierendenkohorten „veralten“ und dadurch unbrauchbar werden.

Die von den Studierenden eingereichten Lösungen werden im hier betrachteten Szenario automatisch oder teilweise automatisch bewertet. Ein sog. Autobewerter (Grader) untersucht eingereichte Lösungen hinsichtlich korrekter Funktion und hinsichtlich weiterer Qualitätsaspekte. Die Konzipierung und Realisierung automatisiert bewertbarer Programmieraufgaben ist zu aufwändig, als dass vielen (>50) Studierenden je eine eigenständig konzipierte Aufgabe zugeteilt werden könnte. Außerdem ist es schwierig, mehrere Aufgaben so zu konzipieren, dass sie zwar verschieden sind, aber dennoch die gleichen Lernziele fördern und vom gewünschten Schwierigkeitsgrad sind. Aus diesem Grunde kann es sinnvoll sein, bestehende Aufgaben durch Einführung von sog. Variationspunkten zu individualisieren. Aus einer Beispielaufgabe wird so eine Aufgabenschablone, die auf studentische Anforderung hin automatisch durch konkrete, häufig zufällig gewählte Variationspunktwerte instanziiert wird.

Häufig sind Autobewerter Backend-Systeme, die nicht direkt mit der Lehrkraft und mit den Studierenden kommunizieren. In der Regel als Frontend eingesetzte Lernmanagementsysteme (LMS) sollen in der Lage sein, Wertebelegungen der Variationspunkte einer Aufgabe entweder automatisch oder in Interaktion mit dem Benutzer vorzunehmen. Dies soll im Sinne der Wiederverwendbarkeit für viele an das LMS angebundene Autobewerter möglichst in einer Form erfolgen, die unabhängig von den Erfordernissen eines bestimmten Autobewerters ist.

1.2 Forschungsfrage

Variationspunkte und deren Wertemengen müssen vom Aufgabenautor einer automatisiert bewerteten Programmieraufgabe spezifiziert werden. Schon bei sehr einfachen Aufgaben bestehen Abhängigkeiten zwischen den einzelnen Variationspunkten, die bei der Wahl der konkreten Werte zu berücksichtigen sind. Der Aufgabenautor steht vor dem Problem, die Abhängigkeiten so zu spezifizieren, dass die Spezifikation einerseits kompakt und verständlich ist, und dass diese andererseits eindeutig ist, so dass die beteiligten Systeme effizient und automatisch korrekte, den Abhängigkeiten gehorchende Wertekombinationen auswählen können.

Durch Kombinationseffekte kann die Menge aller gültigen Wertbelegungen bereits bei einfachen Aufgaben so groß sein, dass eine Auflistung aller gültigen Belegungen an Ressourcengrenzen scheitert. Es wird daher eine Spezifikation von Variationspunkten und deren Wertemengen und Abhängigkeiten gesucht, die verschiedene Systeme (LMS, Autobewerter, ggf. eine zwischen diesen vermittelnde Middleware) in verschiedenen technischen Ökosystemen in die Lage versetzt, ausgewählte Wertbelegungen einer sehr großen Wertemenge zu generieren, deren Abhängigkeiten zu prüfen und ggf. bestimmte Wertbelegungen in einem benutzerfreundlichen Dialog auszuwählen.

1.3 Forschungsmethode

Methodisch schlagen wir anhand einer Beispielaufgabe, die mehrere voneinander abhängende Variationspunkte besitzt, eine Notation im XML-Format vor. Wir veranschaulichen das Format als AND-XOR-Baum. Wir betrachten die an der Erstellung der Aufgabe und der Bewertung von Einreichungen beteiligten Personen, Artefakte und Prozesse und analysieren, inwiefern das vorgeschlagene Format geeignet ist, die jeweiligen Anforderungen zu erfüllen. Um mögliche Anforderungen eines Graders zu illustrieren, beziehen wir uns beispielgebend auf den Grader Graja. Wir realisieren einen Benutzerdialog zur manuellen Auswahl konkreter Werte und untersuchen, inwiefern die vorgeschlagenen Datenstrukturen geeignet sind, alle anfallenden manuellen und automatisierten Aufgaben effizient zu bewältigen.

Wir beschränken uns in diesem Aufsatz auf Variationspunkte mit endlichen Mengen vorzugebender diskreter Werte und auf kontinuierliche Wertebereiche, die durch eine vorzugebende Anzahl von Samples diskretisiert werden.

Wir beschäftigen uns insbesondere mit der Frage, wie lange Auflistungen gültiger Wertkombinationen durch die Angabe von Ableitungsvorschriften ersetzt werden können. Ein besonderes Augenmerk legen wir auf eine redundanzfreie Beschreibung der Variantenmenge in einer Notationssprache, die in allen beteiligten Systemen zur Verfügung stehen kann.

Wir beschränken uns auf die ausführliche Betrachtung einer Beispielaufgabe. Nicht Bestandteil dieses Aufsatzes sind Beschreibungen, wie der Vorschlag auf weitere Aufgaben ausgedehnt werden kann. Nichtsdestotrotz wurden im Rahmen der Entwicklung des Vorschlages im Sinne eines proof of concept mehrere Aufgaben aus einem bestehenden Java-Aufgabenpool in Schablonen konvertiert, um die grundsätzliche Eignung des Vorschlags für weitere Aufgabentypen abzusichern.

1.4 Begriffe

Zunächst definieren wir einige Begriffe. Eine *Aufgabenschablone* (*task template*, kurz *tt*) besitzt variabel gestaltete Stellen. Setzt man an diesen Stellen konkrete Werte ein, entsteht eine *Aufgabeninstanz* (*task instance*, kurz *ti*).

Die Begriffe Schablone und Instanz werden in der verwandten Disziplin der Produktlinienentwicklung (vgl. Abschnitt 1.5) selten genutzt. Die Instanziierung wird dort in zwei Schritte unterteilt: zuerst werden Variablen mit konkreten Werten belegt (*resolution* [12] bzw. *binding* [15]) und anschließend werden variable Artefakte in konkrete Artefakte überführt (*materialization* [12] bzw. *realization* [15]). Statt von Schablonen spricht man in der Produktlinienentwicklung zumindest bei Nutzung orthogonaler Variabilitätsmodelle davon, ein Produktmodell durch ein zusätzliches Variabilitätsmodell zu ergänzen.

Ein *variation point* (dt. *Variationspunkt*, kurz *Vp*, vgl. [15]) wird durch den Namen und den Datentyp einer variabel gestalteten Stelle in einer Programmieraufgabe umschrieben. Eine *variant* (dt. *Variante*, kurz *V*) ist ein an einer variablen Stelle eingesetzter, konkreter Wert. Diese Begriffe haben sich im Bereich der Produktlinienentwicklung eingebürgert.

Eine *composite variant* (kurz *CV*) enthält für jeden Variationspunkt einer Aufgabe einen konkreten Wert, zusammengefasst zu einem Tupel. Ein *composite variation point* (kurz *CVp*)

beschreibt das Tupel aller Variationspunkte einer Aufgabe. Mit *Constraint* bezeichnen wir eine Bedingung, die die Wahl von konkreten Werten für die einzelnen Variationspunkte einschränkt. Die *composite variant set* (kurz *CVs*) einer Programmieraufgabe ist die Menge aller Varianten aller Variationspunkte der Aufgabe, deren Wertkombinationen „sinnvoll“ sind, d. h. deren gewählte Werte inhaltlich konsistent sind und den Constraints genügen.

Die Variantenmenge eines Variationspunktes wird durch eine *variation specification* (kurz *VSpec*) beschrieben (vgl. [12]). Die Spezifikation aller Variationspunkte einer Aufgabe ist eine *composite variation specification* (kurz *CVSpec*). Will man eine Aufgabenschablone instanziieren, muss man jede einzelne VSpec *auflösen* (*resolve*). Der Prozess der *variation resolution* (kurz *Vr*) greift dabei auf eine Referenz auf den aufzulösenden Variationspunkt und dessen gewählte Variante zurück. Eine *composite variation resolution* (*CVr*) beschreibt den Auflösungsprozess für alle Variationspunkte der Aufgabe.

1.5 Verwandte Arbeiten

1.5.1 Nutzen individueller Aufgaben

Individuelle Aufgaben ermöglichen verschiedene Anwendungen. Zum einen betrachten wir Aufgabenvarianten, die dasselbe zu lernende Konzept betreffen und dabei dennoch von unterschiedlichem Schwierigkeitsgrad sind. Liegen solche Aufgabenvarianten vor, kann man diese gezielt anhand des Lernstandes an Studierende ausgeben. Andererseits kann es sinnvoll sein, mehrere Aufgabenvarianten gleichen Schwierigkeitsgrades an denselben Studenten auszugeben, wenn dieser ein Konzept durch intensives Üben verinnerlichen möchte.

Ein weiterer wichtiger Nutzen von variablen Programmieraufgaben ist die Eindämmung von wenig lernförderlichen Plagiaten. Die offensichtliche Alternative, Software zur Plagiaterkennung einzusetzen (z. B. JPlag [1]), scheitert in der Regel bei Anfängeraufgaben, weil die Einreichungen häufig zu kurz sind, um Plagiate sicher feststellen zu können. Zudem stellt sich die Frage, was der Lernmotivation zuträglicher ist: die Aussicht auf Bestrafung eines Plagiats oder der Ansporn, eine individuelle Aufgabe zu lösen. Beispielhaft sei eine Studie [2] zitiert, die drei Hauptfaktoren und Gründe für nicht-plagiierendes Verhalten nennt (s. Abbildung 1). Der Einsatz einer Software zur Plagiatserkennung unterstützt die Gründe 5 und 6 und trägt so zur Eindämmung von Plagiaten bei. Individuelle Aufgaben können die Gründe 1, 2 und 8 stützen und dadurch einen wichtigen Beitrag zur Eindämmung von Plagiaten leisten.

	Reason	Factor 1	Factor 2	Factor 3
1.	Want to know what your work is worth	.772		
2.	Pride in your work	.671		
3.	Can get good marks without cheating	.602		
4.	Against your moral values	.442		
5.	Penalties if caught are too high		.822	
6.	Fear of being found out		.684	
7.	Never thought about it			.903
8.	Don't know how to			.585
	Eigenvalues	1.77	1.68	1.32
	Percentage of variance	17.7	16.8	13.2

Factor 1: Personal integrity.
Factor 2: Fear.
Factor 3: Never considered.

Abbildung 1: Reasons for not cheating: rotated factor matrix (entnommen aus [2])

1.5.2 Individualisierbare Aufgaben in anderen Fachgebieten

Insbesondere in den Fachgebieten Mathematik und Physik werden individualisierbare Aufgaben schon seit längerer Zeit eingesetzt (s. etwa [3]).

Mathematikaufgaben lassen sich häufig sehr leicht variieren, indem andere Werte eingesetzt werden. Ein Beispiel ist in Tabelle 1 gezeigt.

Zeile	Beschreibung	Aufgabe	Musterlösung
1	**Beispielaufgabe**	**Berechnen Sie** $\log_4 \sqrt[6]{2}$**. Geben Sie den Lösungsweg an.**	$x = \log_4 \sqrt[6]{2}$ $\Leftrightarrow 4^x = \sqrt[6]{2}$ $\Leftrightarrow 2^{2x} = 2^{\frac{1}{6}}$ $\Leftrightarrow 2x = \frac{1}{6}$ $\Leftrightarrow x = \frac{1}{12}$
2	**Aufgabe mit Variationspunkten**	**Berechnen Sie** $\log_{p^2} \sqrt[q]{p}$**. Geben Sie den Lösungsweg an.** wobei $p, q \in N, p \geq 2, q \geq 3$. Für eine vergleichbare Schwierigkeit wird q=2 ausgeschlossen, da angenommen wird, dass das Verständnis der Quadratwurzel weniger schwer ist als das Verständnis der allgemeinen Wurzel.	$x = \log_{p^2} \sqrt[q]{p}$ $\Leftrightarrow (p^2)^x = \sqrt[q]{p}$ $\Leftrightarrow p^{2x} = p^{\frac{1}{q}}$ $\Leftrightarrow 2x = \frac{1}{q}$ $\Leftrightarrow x = \frac{1}{2q}$
3	**Variante für** $\boldsymbol{p = 3, q = 5}$	**Berechnen Sie** $\log_9 \sqrt[5]{3}$**. Geben Sie den Lösungsweg an.**	$x = \log_9 \sqrt[5]{3}$ $\Leftrightarrow 9^x = \sqrt[5]{3}$ $\Leftrightarrow 3^{2x} = 3^{\frac{1}{5}}$ $\Leftrightarrow 2x = \frac{1}{5}$ $\Leftrightarrow x = \frac{1}{10}$

Tabelle 1: Mathematikaufgabe mit Variante

Die Aufgabe mit Variationspunkten in Zeile 2 (parametriert durch zwei Parameter p und q) ist eine Verallgemeinerung der konkreten Ausgangsaufgabe (Zeile 1) – eine sog. Aufgaben*schablone*. Die Schablone wird keinen Studierenden zur Verfügung gestellt, sondern stattdessen erhalten Studierende davon abgeleitete Aufgaben*instanzen*. Eine Instanz für p=3, q=5 ist in der Tabelle in Zeile 3 dargestellt.

Kashy et. al. [3] berichten von einer Evaluierung des Effekts individueller Physikaufgaben auf den Lerneffekt bei den Studierenden. Bereits einfache Rechenaufgaben, bei denen Studierende eine Formel auf gegebene Zahlwerte anwenden müssen, führen zu einer positiven Korrelation zwischen Aufgabenerfolg und Erfolg in der abschließenden Prüfung, wenn die Zahlwerte individualisiert werden. In weitaus höherem Maße kann eine solche Korrelation bei komplexeren „Applet"-Aufgaben beobachtet werden. Diese fordern von Studierenden ein tieferes Problemverständnis und von Aufgabenerstellern ein komplexeres Design mit Wechselwirkungen der gewählten „Stellen" zwischen verschiedenen Artefakten (Funktionsgraph, Eingabegrößen, Antwortoptionen).

1.5.3 Individualisierbare Programmieraufgaben

Brusilovsky et. al. berichten über das QuizPACK-System für parametrisierte Aufgaben für die Programmiersprache C [4]. Mit QuizJET wurde die Idee später auf Java-

Programmieraufgaben übertragen [5]. Die Autoren betrachten einen ganz bestimmten Aufgabentyp („parameterized code-execution exercises“), bei dem Studierende ein gegebenes Programm durchdenken müssen und am Ende eine Frage wie „Welchen Wert hat die Variable x?“ beantworten müssen. Der mit Variationspunkten versehene Programmtext wird von QuizPACK individualisiert. Die eingereichte Antwort wird mit dem Ergebnis verglichen, das QuizPACK automatisch durch Ausführung des Programmtextes berechnet. In der Bloom’schen Lernzieltaxonomie fördern „code-execution“-Aufgaben niedrigere Lernziele (Anwenden, Analyse) als Aufgaben, bei denen Studierende selbst Programmcode schreiben müssen (Synthese). Der letztgenannte Aufgabentyp wird von den Systemen QuizPACK und QuizJET nicht unterstützt.

Radosevic et. al. beschreiben in [6] ein sehr einfaches System zur Generierung von Varianten für C++-Programmieraufgaben. Es werden einfache Schlüssel-Wert-Paare für „Eigenschaften“ des Programms eingesetzt. Als Werte dienen einfache Zeichenketten. Es gibt keine Vorkehrungen zur Gruppierung zusammengehöriger Parameter. Artefakte (vorgegebener Programmtext, Aufgabentext) werden bewusst nicht voneinander getrennt. Der Aufgabentext steht als Kommentar im Programmtext. Zur Identifizierung von Varianten wird die Matrikelnummer genutzt.

Spinellis et. al. stellen das System Jarpeb zur automatisierten Bewertung von Java-Programmen vor [7]. Wertebereiche für individualisierte Bezeichner (Klassen, Methoden, etc.) werden zufällig aus einem Dictionary bedeutungsloser Zeichenketten gebildet. Dem Bericht ist nicht zu entnehmen, inwiefern Aufgabentypen, Artefakte, Stellentypen, etc. strukturiert werden. Es werden die Effekte des Jarpeb-Einsatzes aus Sicht der Studierenden genannt: reduzierte Plagiierung, besseres Verstehen der Übungen, bessere Unterstützung des Lernens, erhöhte wahrgenommene Fairness, vermehrter Spaß und Interesse an den Übungen, aber auch erhöhte wahrgenommene Schwierigkeit der Übungen.

Ein Ansatz zur Generierung individueller Aufgaben für ein Teilgebiet der Programmierung (Auswertung von Ausdrücken) wird in [8] diskutiert. Bei der Generierung von Ausdrücken werden Regeln berücksichtigt, die die Qualität der erzeugten Ausdrücke verbessern. Bei einem so eingeschränkten Aufgabentyp können die Regeln sehr spezifisch sein.

In letzter Zeit wird verstärkt über Bemühungen berichtet, Programmieraufgaben automatisch auf der Grundlage von Ontologien zu erzeugen. Ein früher Ansatz für die Datenbankabfragesprache SQL wird in [9] besprochen. Schon seit längerer Zeit wird die automatische Generierung von MCQ[2]-Tests beforscht [10], ein Verfahren, das prinzipiell auch auf die Programmierdomäne angewendet werden kann, aber dessen generierte Fragen in der Regel nicht auf Lernziele der oberen Bloom’schen Taxonomie-Ebenen abzielen.

1.5.4 Feature models

In der Produktlinienentwicklung wurden und werden sogenannte feature models [11] genutzt, um Varianten und deren Randbedingungen darzustellen. Die Domäne der Produktlinienentwicklung hat mit der Domäne der individualisierbaren Programmieraufgaben gemeinsam, dass Variationspunkte, Wertausprägungen und teilweise komplexe Randbedingungen formuliert werden müssen. Seit 1990 gab es eine Fülle von Erweiterungen der ursprünglichen Modellierungsansätze.

Beispielhaft sei die OMG Revised Submission zur Common Variability Language (CVL) genannt [12], in der Verbindungen zwischen dem Variabilitätsmodell und dem eigentlichen Produktmodell die Auswirkungen von Wertbelegungen (dort *resolution* genannt) spezifizieren. Die Spezifikationsmöglichkeiten von Sprachen wir der CVL gehen weit über die Erfordernisse einer variablen Programmieraufgabe hinaus. Das verwundert nicht, da in der Produktlinienentwicklung Modelle mit hunderten von Variationspunkten keine Seltenheit sind, während wir in variablen Programmieraufgaben kaum mit mehr als einer kleinen zweistelligen Anzahl von Variationspunkten rechnen. Außerdem ist in variablen Programmieraufgaben nach unserer Einschätzung die sog. *variability realization* [20] häufig

[2] MCQ = multiple choice question

einfach, z. B. indem die variablen Stellen in einem Artefakt automatisch gesucht und durch konkrete Werte ersetzt werden.

Nichtsdestotrotz sind viele Konzepte der Produktlinienentwicklung auf variable Programmieraufgaben übertragbar. Ein Beispiel: Das CVL-Konzept der VSpec derivations ist zumindest in Teilen vergleichbar mit dem in diesem Aufsatz vorgeschlagenen Konzept der Ableitungsvorschriften. Häufig ist es in individualisierbaren Programmieraufgaben möglich, die gültigen Varianten eines Variationspunktes automatisch von konkret gewählten Varianten anderer Variationspunkte abzuleiten, anstatt umständlich entweder alle verbotenen Varianten-Kombinationen oder alle gültigen Varianten-Kombinationen aufzulisten und Äquivalenz- und Implikationsbeziehungen zu formulieren.

Wir werden im weiteren Verlauf an den Stellen, an denen wir weitere Ähnlichkeiten zu Ansätzen der Produktlinienentwicklung erkennen, auf entsprechende Literatur referenzieren.

2 Anforderungen an eine Notation

Eine Notation zur Spezifikation der Variantenmenge einer Programmieraufgabe sollte folgenden Anforderungen genügen. Die kursiv gesetzten Anforderungen werden von der in diesem Aufsatz vorgeschlagenen Notation noch nicht unterstützt. Alle anderen Anforderungen können im Prinzip unterstützt werden, wenn dies auch nicht immer ausdrücklich in diesem Aufsatz ausgeführt wird.

(1) Mehrdimensional und beschränkbar. Die Notation muss mehrere Variationspunkte in einer Aufgabe abdecken können. Constraints (Beschränkungen) zwischen den Variationspunkten müssen beschreibbar sein, so dass gültige von nicht gültigen Varianten der Variationspunkte unterschieden werden können.

(2) Offen für beliebige Werttypen. Jeder Variationspunkt hat in der Regel seinen eigenen Datentyp. Sowohl einfache Datentypen (Zahl, Zeichenkette, Zeichen) als auch komplexere Datentypen (Datei, Programmfragment, Beschreibungsfragment, Programmbibliothek, Datenbanktabelle, Internetadresse, Port, Algorithmus, etc.) sollen abbildbar sein. Es sollten sowohl endliche Variantenmengen als auch prinzipiell unendliche Variantenmengen (z. B. kontinuierliche Intervalle) unterstützt werden.

(3) Kompakt und redundanzarm. Die Notation sollte kompakter sein als die Auflistung aller den Beschränkungen gehorchenden Wertkombinationen der einzelnen Variationspunkte. Wiederholungen von identischen Wertangaben bei der Beschreibung der Variantenmenge sollen aus Gründen der Wartbarkeit weitgehend vermieden werden.

(4) Programmiersprachen- und Grader-unabhängig. Da ein LMS, das variable Programmieraufgaben unterstützt, in der Lage sein soll, die Notation zu interpretieren, sollte die Notation nicht nur für eine bestimmte Programmiersprache oder einen bestimmten Grader entworfen sein.

(5) Eindeutig und maschinenlesbar. Die Notation kann als Grundlage dafür dienen, automatisch und zufallsbasiert eine Variante einer Aufgabe auszuwählen. Dazu muss die Spezifikation automatisch aufgelöst werden, d. h. für jeden Variationspunkt muss automatisch eine konkrete Variante gewählt werden. Damit dies funktioniert, muss die Variantenmenge eindeutig und maschinenlesbar formuliert sein.

(6) Universell und adaptiv. Die Notation soll in der Lage sein, jede denkbare Variationssituation beschreiben zu können. Im äußersten Fall bei sehr vielen Constraints z. B. dadurch, dass sie jede gültige Wertkombination auflistet. Sie soll adaptiv sein, d. h. sie soll nur dann ausführlich sein, wenn es die vorliegende Variationssituation erfordert. Beispielsweise soll bei Fehlen jeglicher Constraints eine sehr kompakte Darstellung möglich sein.

(7) Durch Menschen lesbar und bearbeitbar. Die Variantenmenge muss durch einen Aufgabenautor beschrieben werden und daher durch Menschen lesbar und bearbeitbar sein. Eine Aufgabe soll aus Studierendensicht Realitätsbezug besitzen. Dies schließt z. B. die Bildung von Zufallszeichenketten als alleinigen Generierungsmechanismus für Variationspunkte aus.

(8) Generierbar. Das Bearbeitungsformat ist nicht notwendigerweise dasselbe wie das Notationsformat. Es ist denkbar, dass der Aufgabenautor die Variantenmenge zunächst in einer eigenen, Programmiersprachen- und Grader-abhängigen Notation erstellt. Hieraus soll es möglich sein, ein den anderen Anforderungen gehorchendes, neues, die Variantenmenge beschreibendes Artefakt zu generieren.

(9) Bevorzugte Werte: Die Notation muss es erlauben, bevorzugte Werte oder Wertkombinationen anzugeben.

(10) Freie Parameter: Die Notation muss es erlauben, die Variationspunkte, deren Werte frei wählbar sind, von den Variationspunkten, deren Werte von anderen Werten abhängig sind, zu unterscheiden.

(11) Schwierigkeitsgrad: Die Notation muss es erlauben, zu jeder Wertkombination eine Aufgabenschwierigkeit anzugeben.

3 Elemente einer individualisierbaren Programmieraufgabe

3.1 Beispielaufgabe

Gegeben sei eine sehr einfache Java-Programmieraufgabe:

> *Write a `main` method in class `Letters` in the default package that reads a series of characters from user input. Your program should output a statistic of letters in the input, i. e. the percentage of characters in { A-Z }. Example (user input is **highlighted**):*
>
> ```
> Give me characters, please: x.ÜL:0€ÁHDú/7Ú
> 21.43 % are letters
> ```
>
> *Your program should print 2 decimal places of the percentage value. Lower case letters should not be counted.*

Aufgabe 1: Einfache Beispielaufgabe

Wir betrachten neben dem Aufgabentext auch die weiteren Artefakte der Aufgabe. Wir nutzen dazu beispielgebend den Autobewerter Graja [13]. Dieser definiert durchzuführende Tests als JUnit-Testmethoden. Zudem werden Randbedingungen und weitere Begleitinformationen einer Einreichung in <test-configuration>-Elementen der ProFormA-taskxml-Datei [14] formuliert.

Der Graja-JUnit-Bewertungscode könnte wie folgt aussehen (vgl. Quelltext 1). Eine JUnit-Testmethode shouldWorkWithDefaultInput erzeugt zunächst die erwartete Ausgabe für eine Default-Eingabe. Dazu bedient sie sich einer Helperklasse, in der die Musterlösung umgesetzt ist. Anschließend ruft die Testmethode die studentische Einreichung auf (Support.callMainAndReturnOutput), vergleicht die beiden Ergebnisse (compareStrings) und stellt das Vergleichsergebnis tabellarisch dar. Wir haben hier aus Platzgründen lediglich einen Test implementiert. In einer realen Aufgabe würde man diesen Test um weitere Tests mit abweichenden Benutzereingaben ergänzen.

```
public class Helper {
    static final String DEFAULT_USER_INPUT= "x.ÜL:0€ÁHDú/7Ú";
    static String getPercentage(String input) {
        Matcher matcher= Pattern.compile("[A-Z]").matcher(input);
        int count= 0;
        while (matcher.find()) count++;
        return String.format("%.2f", (double)count*100/input.length());
    }
}
```

```
public class Grader extends AssignmentGrader {
    @Test public void shouldWorkWithDefaultInput() {
        String expected;
        try (Formatter f= new Formatter(new StringBuilder())) {
            f.format("Give me characters, please: %s%n", Helper.DEFAULT_USER_INPUT);
            f.format("%s %% are letters", Helper.getPercentage(Helper.DEFAULT_USER_INPUT));
            expected= f.toString();
        }
        Class<?> submission= getPublicClassForName("Letters");
        String observed= Support.callMainAndReturnOutput(
                                            submission, new String[]{}, Helper.DEFAULT_USER_INPUT);
        DiffHelper.compareStrings(expected, observed)
            .normalizeOutput(
                      new StringNormalizer(StringNormalizer.StandardRule.IGNORE_NEWLINE_DIFFERENCES))
            .start();
    }
}
```

Quelltext 1

Die taskxml-Datei der Aufgabe enthält neben dem Aufgabentext und dem obigen Testcode weitere Konfigurationselemente. Einerseits ist hier die folgende Randbedingung zu nennen, mit der Graja für eine Einreichung prüft, ob diese nur die erwartete Datei enthält:

```
    <graja:submissionRules>
        <graja:restrictFile name="Letters.java"/>
    </graja:submissionRules>
```

Quelltext 2

Zusätzlich sind in der taskxml-Datei weitere Testkonfigurationen und Bewertungsinformationen enthalten:

```
    <p:test id="method_0">
      <p:title>Should output 21.43 for default input</p:title>
      <p:test-type>graja-submodule-junit-method</p:test-type>
      <p:test-configuration>
        <g:junitSubmoduleMethodCfg
                    method="de.hsh.charstatv00.grader.Grader#shouldWorkWithDefaultInput/>
      </p:test-configuration>
    </p:test>
...
        <grp:test-group testref-id="category_1" score-max="4.0">
          <grp:test-group-members>
            <grp:test-element testref-id="method_0" score-max="4.0">
              <grp:description>For the default input given in the assignment's text your
program "Letters" should output the percentage '21.43'.</grp:description>
            </grp:test-element>
          </grp:test-group-members>
        </grp:test-group>
```

Quelltext 3

3.2 Aufgabenschablone

Die soeben beschriebene Aufgabe besitzt keine Variationspunkte. Sie ist eine Aufgaben*instanz*. Es besteht jedoch Potential, in diese Aufgabe Variationspunkte einzubauen. Eine mögliche Aufgabenvariante wäre die folgende:

*Write a `main` method in class `Digits` in the default package that reads a series of characters from user input. Your program should output a statistic of numbers in the input, i. e. the percentage of characters in { 0-9 }. Example (user input is **highlighted**):*

```
Give me characters, please: x.ÜL:0€ÁHDú/7Ú
14.285714 % are numbers
```

Your program should print 6 decimal places of the percentage value.

Aufgabe 2: Variante der Aufgabe 1

In der vorstehenden Aufgabenvariante wurden die veränderten Passagen grau unterlegt. Die neue Variante ist strukturähnlich zur ersten Aufgabe, betrifft aber eine etwas andere Fachdomäne. Es sollen Ziffern statt Buchstaben gezählt werden. Dementsprechend wurde der Name der zu erstellenden Klasse geändert. Außerdem wird in der neuen Variante eine höhere Anzahl von Dezimalstellen gefordert. Zu guter Letzt wurde der letzte Satz „*Lower case letters should not be counted.*“ gestrichen, weil er im Kontext dieser Variante unsinnig ist.

Eine verallgemeinerte Aufgabenstellung, die beide bisherigen Aufgabenvarianten gleichermaßen abdeckt, soll nun als Aufgabenschablone formuliert werden:

*Write a `main` method in class `%vp{className}` in the default package that reads a series of characters from user input. Your program should output a statistic of %vp{domain} in the input, i. e. the percentage of characters in { %vp{set} }. Example (user input is **highlighted**):*

```
Give me characters, please: %vp{input}
%vp{output} % are %vp{domain}
```

Your program should print %vp{precision} decimal places of the percentage value. %vp{lowerSentence}

Aufgabe 3: Aufgabentext der Aufgabenschablone

Die hervorgehobenen Stellen im Aufgabentext definieren sieben Variationspunkte. Einige der eingeführten Variationspunkte sind geeignet, die Schwierigkeit der Aufgabe leicht zu erhöhen. Andere dienen lediglich der möglichst umfangreichen Variantenbildung bei gleichbleibender Schwierigkeit. Die folgende Tabelle gibt einen Überblick über alle in der Aufgabenschablone eingeführten Variationspunkte und darüber hinaus über einen weiteren „versteckten“ achten Variationspunkt, der sich zwar nicht im Aufgabentext, aber in weiteren Artefakten wiederfinden wird.

Wir können die acht Variationspunkte gedanklich zu einem Vektor mit acht Komponenten zusammenfassen. Einen solchen Vektor von Variationspunkten nennen wir CVp (vgl. Abschnitt 1.4). Eine individualisierbaren Aufgabe wird charakterisiert durch solch ein CVp, aus dessen Menge sinnvoller Werte schließlich die Aufgabenvarianten generiert werden können.

Nr.	Name	Datentyp	Sinnvolle Werte
1	input	String	Der beispielhaft eingegebene Text wie er im Aufgabentext abgedruckt ist. Dieser wird nun variabel gestaltet, weil dieser je nach Ausprägung der anderen Variationspunkte mit variiert werden könnte[3].
2	domain	String	Name der Anwendungsdomäne aus der Menge { „letters“, „numbers“ }. Mit diesem Vp unterscheiden wir also zwischen den beiden oben abgedruckten Aufgabeninstanzen.
3	class-Name	String	Innerhalb einer domain kann es verschiedene sinnvolle Bezeichner für die von Studierenden zu schreibende Klasse geben. Für die domain „letters“ soll "Letters" als sinnvoller Name gelten. Für die domain „numbers“ sehen wir "Digits", "Numbers", „Figures“ als mögliche realitätsbezogene Bezeichner vor. Wir haben es hier mit einem Vp zu tun, dessen Wertemenge von der Ausprägung eines anderen Vp abhängt.
4	set	String	Auch die Wertemenge dieses Vp hängt von der domain ab. Sie bezeichnet die Menge der zu zählenden Zeichen. Hier sind auch mehrere Varianten denkbar. Bspw. für die domain „letters“ ist set=„A-Z“ eine einfache Variante, während set=„A-Z, Ä, Ö, Ü“ schon etwas aufwändiger in der Umsetzung ist. Eine weitere Ausprägung set=„A-Z, Œ“ erhöht die Schwierigkeit noch einmal, weil hier in der Regel auf einem deutschen Rechner Codierungsprobleme bewältigt werden müssen. Für die domain „numbers“ wird man z. B. den Werte „0-9“ für set vorgeben oder den etwas schwierigeren Wert „0-9, I, V, X, L, C, D, M“, der auch römische Ziffern beinhaltet.
5	count-Lower	Boolean	Falls der Wert dieses Vp true ist, dann soll das eingereichte studentische Programm Kleinbuchstaben mitzählen. Falls false, dann nicht. Denkbar ist auch der Wert null für set=„0-9“. In diesem Fall ist die Anforderung, Kleinbuchstaben mitzuzählen, unsinnig[4]. Die gültigen Werte sind also { null, false, true }. Der in diesem Vp gewählte Wert schlägt sich nur mittelbar im Aufgabentext nieder. Im JUnit-Testcode wird der Wert dieses Vp hingegen direkt verwendet werden.
6	lower-Sentence	String	Der am Ende des Aufgabentextes auszugebende Satz. Hängt ab von der Ausprägung von countLower. Falls countLower=false, dann ist lowerSentence=„Lower case letters should not be counted.” Falls countLower=true ist lowerSetence=„Lower case letters should be counted.” Und falls countLower=null ist lowerSentence=””.
7	precision	Integer	Anzahl Nachkommastellen. Mit diesem Vp erreichen wir leicht eine Verneunfachung der Aufgabenvarianten, ohne dass sich dabei an der Aufgabenschwierigkeit etwas ändert.
8	output	String	Die erwartete Ausgabe wie sie im Aufgabentext steht. Wenn man alle vorstehenden Vp gewählt hat, besitzt output genau einen dazugehörigen sinnvollen Wert, nämlich die erwartete Ausgabe der geforderten Programmvariante.

Tabelle 2: Variationspunkte der Aufgabe 3

3.3 Versuch: Darstellung in einem Featurediagramm

Die acht Variationspunkte besitzen Wertemengen, die teilweise voneinander abhängen. Wir stellen diese beispielhaft durch ein Featurediagramm dar (vgl. Abbildung 3, Legende in

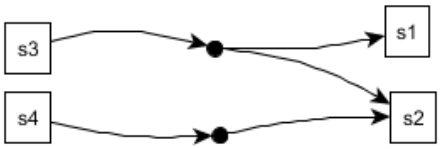
[3] Tatsächlich verzichten wir hier auf eine Variierung, d. h. der Wert dieser ersten Variable wird einer einelementigen Menge entnommen. Dies jedoch nur aus Gründen der knapperen Darstellung in diesem Aufsatz. In einer realen Aufgabe würden wir auch die Variable input echt variabel gestalten.

[4] Die in [15] beschriebene Abhängigkeit vom Typ „Variant excludes variation point“ ist hiermit vergleichbar.

Abbildung 2). Vom Wurzelknoten in der Mitte nach außen gelesen werden Variationspunkte und deren Wertebereiche als XOR-Hyperkanten dargestellt. Bspw. verweist *(root)* über einer Hyperkante mit „Tail“-Kardinalität 1 und „Head“-Kardinalität 5 nach unten auf fünf verschiedene Ausprägungen des Variationspunkts *set*.

Abhängigkeiten der Variationspunkte untereinander werden ebenfalls als XOR-Hyperkanten dargestellt. Diese sog. „requires“-Beziehungen verlaufen „cross-tree“, so dass am Ende kein Baum, sondern ein Graph entsteht. Wir haben die „requires“-Beziehungen der Übersichtlichkeit halber nur von Variationspunkten mit kleineren Nummern (im Sinne der Nummerierung in Tabelle 2) zu Variationspunkten mit größeren Nummern dargestellt. Wir gehen also davon aus, dass die Variablenbelegung stets in der Reihenfolge 1 bis 8 erfolgt. Wenn bspw. domain den Wert „letters“ besitzt, dann muss set einen der drei Werte „A-Z“, „A-Z, Ä, Ö, Ü“, „A-Z, Œ“ annehmen. Falls domain hingegen gleich „numbers“ ist, muss set einen der beiden Werte „0-9“, „0-9, I, V, X, L, C, D, M“ annehmen. In diesem Fall gilt die Abhängigkeit zwar auch umgekehrt, d. h. bei vorgegebenem set-Wert lässt sich der einzige gültige domain-Wert direkt ableiten. Wir haben solche rückwärts gerichteten requires-Beziehungen jedoch aus Gründen der Übersichtlichkeit nicht in das Diagramm aufgenommen.

Das auf diese naive Weise erstellte Featurediagramm ist bereits für eine kleine Aufgabe unübersichtlich. Die Darstellung als Graph ist bereits bei Abhängigkeiten von der geringen Komplexität, wie sie in dieser einfachen Aufgabe vorkommen, ungeeignet. Überdies ist das Diagramm noch höchst unvollständig, denn es zeigt lediglich einen kleinen Ausschnitt aller möglichen Werte für den Variationspunkt „output“. Insbesondere ist es unbequem, Wertausprägungen explizit in das Diagramm aufzunehmen, die sich direkt aus anderen Werten ableiten lassen.



Legende: Rechteckige Knoten („feature“) bezeichnen Variationspunkte und deren Werte. Kleine runde Knoten bezeichnen Hyperkanten. Alle Hyperkanten in unseren Beispielen sind XOR-Kanten, d. h. wenn s3 erfüllt ist, muss genau eines der referenzierten Features s1, s2 erfüllt sein.

Abbildung 2: Legende zum Featurediagramm

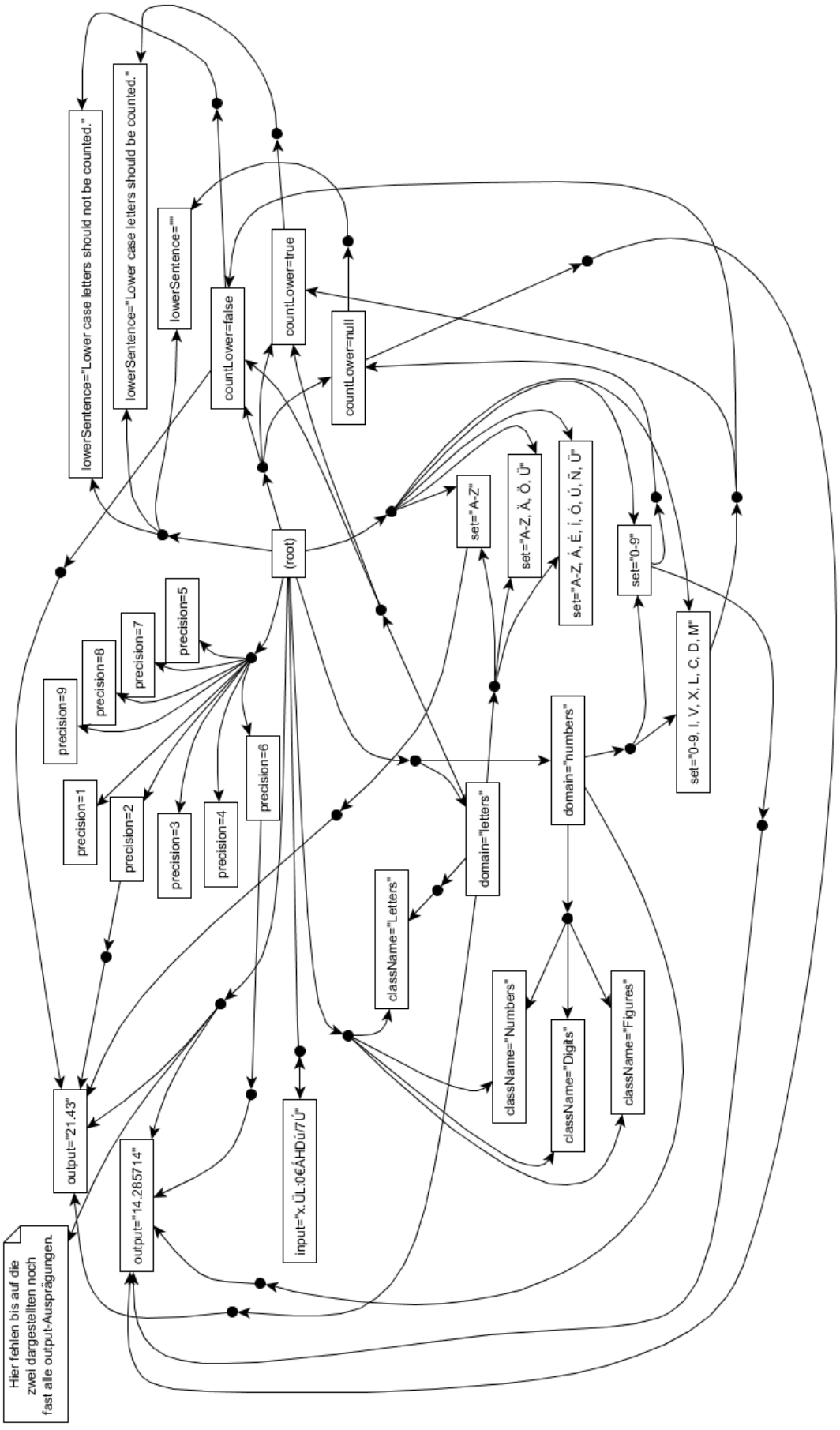


Abbildung 3: Featurediagramm (eigene vereinfachte Notation, ursprünglich orientiert an Laguna, Marques [16])

3.4 Auswirkung der Variationspunkte auf weitere Artefakte der Aufgabe

Von der Einführung der Variationspunkte sind die weiteren Artefakte der Aufgabe betroffen. Wir beginnen mit den Testmethoden. Man kann die Testmethoden dieses Beispiels so generisch formulieren, dass sie die konkrete Ausprägung aller Variationspunkte als Variable nutzt. Wieder am Beispiel von Graja könnte dies wie folgt aussehen. Graja stellt in der Superklasse `AssignmentGrader` eine Methode `getCVr()` zur Verfügung, die für alle Variationspunkte der Aufgabe konkrete Wertbelegungen liefert. Die im folgenden Quelltext grau unterlegten Stellen markieren die im Vergleich zur nicht-variablen Aufgabe geänderten Stellen. Hier ist nun auch die Verwendung der booleschen Variable *countLower* zu erkennen, die ja im Aufgabentext nicht ersichtlich ist:

```
public class Helper {
    static final String DEFAULT_USER_INPUT= "x.ÜL:0€ÁHDú/7Ú";
    static String getPercentage(String input, String set, Boolean countLower, int precision) {
        if (countLower != null && countLower) input= input.toUpperCase();
        Matcher matcher= Pattern.compile("["+set.replace(", ",  "")+"]").matcher(input);
        int count= 0;
        while (matcher.find()) count++;
        return String.format("%."+precision+"f", (double)count*100/input.length());
    }
}
```

```
public class Grader extends AssignmentGrader {
    private CVr cvr;
    @Before public void setup() {
        cvr= getCVr();
    }
    @Test public void shouldWorkWithDefaultInput() {
        String expected;
        try (Formatter f= new Formatter(new StringBuilder())) {
            f.format("Give me characters, please: %s%n", Helper.DEFAULT_USER_INPUT);
            f.format("%s %% are %s", Helper.getPercentage(Helper.DEFAULT_USER_INPUT,
                                        cvr.getString("set"),
                                        cvr.getBoolean("countLower"),
                                        cvr.getInt("precision")),
                                    cvr.getString("domain"));
            expected= f.toString();
        }
        Class<?> submission= getPublicClassForName(cvr.getString("className"));
        String observed= Support.callMainAndReturnOutput(
                                            submission, new String[]{}, Helper.DEFAULT_USER_INPUT);
        DiffHelper.compareStrings(expected, observed)
            .normalizeOutput(
                   new StringNormalizer(StringNormalizer.StandardRule.IGNORE_NEWLINE_DIFFERENCES))
            .start();
    }
}
```

Quelltext 4

Die Wahl einer Variante hat für Graja nicht nur Auswirkungen auf den JUnit-Testcode, sondern auch auf weitere Einstellungen zur Aufgabe. Die Restriktionsregel und weitere Begleitinformationen in der taskxml-Datei müssen nun variabel abhängig von den Variationspunktwerten gestaltet werden. Hierzu verwendet Graja die folgende Notation:

```
    <graja:submissionRules>
        <graja:restrictFile name="%vp{className}.java"/>
    </graja:submissionRules>

    <p:test id="method_0">
      <p:title>Should output %vp{output} for default input</p:title>
      <p:test-type>graja-submodule-junit-method</p:test-type>
      <p:test-configuration>
        <g:junitSubmoduleMethodCfg
                      method="de.hsh.charstatv01.grader.Grader#shouldWorkWithDefaultInput/>
      </p:test-configuration>
    </p:test>
...
        <grp:test-group testref-id="category_1" score-max="4.0">
          <grp:test-group-members>
            <grp:test-element testref-id="method_0" score-max="4.0">
              <grp:description>For the default input given in the assignment's text your
program "%vp{className}" should output the percentage '%vp{output}'.</grp:description>
            </grp:test-element>
          </grp:test-group-members>
        </grp:test-group>
...
```

Quelltext 5

Bündelt man nun die soeben beschriebenen Artefakte, in denen konkrete Werte durch Variationspunkt-Bezeichner ersetzt wurden, so entsteht eine sog. Aufgaben*schablone*. Eine Schablone wird im Falle von Graja als ProFormA-taskxml-Datei bereitgestellt. Diese ist jedoch nur dann zur Bewertung einer Einreichung einsetzbar, wenn zusätzlich eine Wertbelegung der Variationspunkte mitgeliefert wird. Die Schablone muss dann zuerst instanziiert werden, bevor die eigentliche Bewertung beginnen kann. Bei der Instanziierung werden Variationspunkt-Bezeichner automatisch durch die konkreten Werte ersetzt. Im Falle von Graja werden die Zeichenketten der Form `%vp{…}` durch konkrete Werte ersetzt. Der Quellcode der JUnit-Testklasse ist hingegen generisch aufgebaut. Hier ist bei der Instanziierung nichts zu tun. Stattdessen muss die Graja-Laufzeitumgebung dafür sorgen, dass die Methode `getCVr()` den zur Instanz passenden Wertevektor liefert.

Die in einer Aufgabenschablone gespeicherte Variabilitätsinformation wird zu einem bestimmten Zeitpunkt an konkrete Werte „gebunden“ [15] bzw. die Variationspunkte werden „aufgelöst“ („resolved“) [12]. Abhängig davon, wie die Artefakte der Aufgabenschablone gestaltet sind und wie der Grader diese Artefakte verarbeitet, kann die binding time für jedes zur Aufgabe gehörende Artefakt in jede der folgenden Klassen fallen (vgl. auch das *binding time* Konzept der Produktlinienentwicklung etwa in [15]): vor Compilierung, bei der Compilierung, beim Linken, beim Laden oder während der Ausführung.

- Der Aufgabentext wird sinnvollerweise schon vor der Ausführung des Graders „gebunden“, d. h. im Aufgabentext werden schon sehr früh die Platzhalter durch konkrete Werte ersetzt. Die im folgenden Abschnitt beschriebene Instantiation engine des Graders Graja erzeugt den konkreten Aufgabentext beim Erzeugen einer konkreten ProFormA-Datei, was man als Link-Operation deuten kann.
- Ein anderes Artefakt, die JUnit-Tests, erwartet der Grader Graja zur Laufzeit in compilierter Form als Bytecode. Hier führt Graja die Bindung der Variationspunkte an konkrete Werte erst zur Laufzeit des Graders durch.

3.5 Systeme in der Prozesskette

In diesem Abschnitt werfen wir einen Blick auf die technischen Systeme, die an der Auswahl einer Aufgabenvariante und an der Bewertung einer Einreichung beteiligt sind. Ein häufiges Szenario ist ein Lernmanagementsystem (LMS) als Frontend, welches im Hintergrund arbeitende Autobewerter (Grader) steuert. U. u. vermittelt eine Middleware [17] zwischen Frontend und Backend. Das LMS übernimmt normalerweise die Benutzerverwaltung und „kennt“ die handelnden Personen (Lehrkräfte und Studierende). Grader sind meist zustandslose Dienste, die eine gegebene Aufgabe und eine gegebene Einreichung bewerten.

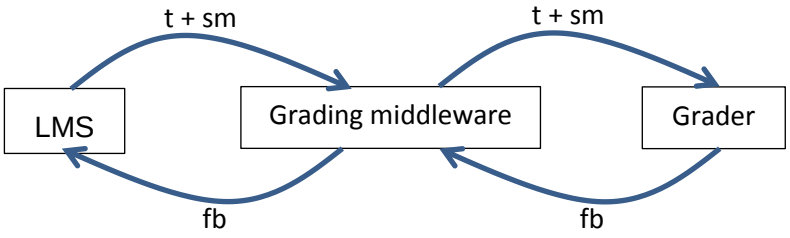


Abbildung 4: Anbindung eines Graders an ein LMS. Verwendete Abkürzungen: t = task (Aufgabe), sm = submission (Einreichung), fb = Feedback.

Um einreichenden Studierenden mehrfache Abgaben zur gleichen Aufgabenvariante zu ermöglichen, muss das LMS sich merken, welche Variante die jeweilige Person bearbeitet. Auch kann dem LMS die Verantwortung zufallen, einer Person eine Aufgabenvariante selbst zuzuweisen. Dafür spricht z. B., dass das LMS die Lernhistorie des Studenten am besten kennt und diese z. B. in die gewählte Schwierigkeit einer Variante einfließen lassen kann.

Andererseits ist der Grader das System mit der besten Kenntnis über technische Besonderheiten der Aufgabe. Hier ist das Wissen über die eingesetzte Programmiersprache, über deren Standarddatentypen, etc. beheimatet.

Bei der Entwicklung der Notation stand folgendes Beispielszenario Pate.

> *Ein LMS bietet der Lehrkraft die Möglichkeit, eine Programmieraufgabe anzulegen und zu konfigurieren. Die Lehrkraft lädt eine ProFormA-taskxml-Datei hoch, in der sich im Sinne von Abschnitt 3.4 eine Aufgabenschablone befindet. Die Lehrkraft testet die Aufgabe, indem sie selbst eine Musterlösung einreicht. Das LMS bietet der Lehrkraft einen Dialog an, in dem die Lehrkraft die Wertbelegung der Variationspunkte manuell so vornimmt, dass sie zu der eingereichten Musterlösung passen. Das LMS erstellt daraufhin mit den von der Lehrkraft eingegebenen Variationspunktwerten eine Aufgabeninstanz und reicht diese zusammen mit der eingereichten Musterlösung an die Middleware bzw. an den Grader. Jener bewertet die Einreichung und liefert das Feedback zurück, welches vom LMS dargestellt wird.*
>
> *Das LMS schaltet die Aufgabe zu einem bestimmten Termin für Studierende frei. Studierende rufen die Aufgabe mit einem Internetbrowser auf. Das LMS zeigt jedem Studierenden eine eigene Variante der Aufgabe an. Dazu erstellt das LMS in dem Moment, in dem ein Student erstmals die Aufgabe anfordert, eine Aufgabeninstanz. Wir gehen zunächst davon aus, dass das LMS einfach eine zufällige Wertbelegung der Variationspunkte vornimmt. Denkbar ist jedoch auch, dass das LMS weitere Faktoren außer dem Zufall in die Werteauswahl einfließen lässt. Auf jeden Fall beachtet das LMS die gegenseitigen Abhängigkeiten der Variationspunkte. Das LMS speichert die für den betreffenden Studenten geltende Wertbelegung und/oder die resultierende Aufgabeninstanz persistent ab. Der eigentliche Einreichungs- und Bewertungsvorgang über die Middleware unterscheidet sich nun nicht mehr von einer „normalen" Aufgabe. Ggf. kann das LMS dem Studenten, wenn er die Aufgabe gelöst hat, anbieten, eine individuelle neue Variante zu Übungszwecken zu generieren.*

In dem Szenario wird die in einer Aufgabenschablone gespeicherte Variabilitätsinformation zu dem Zeitpunkt „gebunden", in dem eine Person eine neue Variante anfordert. Abhängig davon, wie die Artefakte der Aufgabenschablone gestaltet sind und wie der Grader diese Artefakte verarbeitet, kann die binding time in jede der folgenden Klassen fallen: vor Compilierung, bei der Compilierung, beim Linken, beim Laden oder während der Ausführung (vgl. auch das *binding time* Konzept der Produktlinienentwicklung etwa in [15]). Der Aufgabentext wird sinnvollerweise schon vor der Ausführung des Graders „gebunden", d. h. im Aufgabentext werden schon sehr früh die Platzhalter durch konkrete Werte ersetzt. Ein anderes Artefakt, das wir in Abschnitt 3.4 gesehen haben, sind die JUnit-Tests. Bspw.

erwartet der Grader Graja diese Tests in compilierter Form als Bytecode. Unser Vorgehen in Abschnitt 3.4 legt nahe, hier die Bindung erst zur Laufzeit des Graders durchzuführen.

Im oben skizzierten Szenario hat das LMS die alleinige Verantwortung für die Auswahl von Wertbelegungen und Aufgabeninstanziierungen. Das LMS wird diese Aufgaben in den allermeisten Fällen nicht ohne Unterstützung des Graders meistern können. Wir werden zwei Aspekte genauer beleuchten. Zum einen ist es denkbar, dass das LMS es mit Aufgabenartefakten zu tun bekommt, deren Instanziierung Grader-spezifisches Wissen erfordert. Zum anderen ist es denkbar, dass die Menge gültiger Werte für einen Variationspunkt nur dem Grader bekannt ist.

3.5.1 Proprietäre Aufgabenartefakte

Abschnitt 3.4 hat verdeutlicht, dass die von Variationspunkten betroffenen Aufgabenartefakte sehr Grader-spezifisch sein können. Ggf. sind nicht nur Textdateien, sondern auch Dateien in proprietären Binärformaten betroffen. Nur der Grader wird letztverantwortlich in der Lage sein, aus einer Aufgabenschablone eine Aufgabeninstanz zu erzeugen. Einfache Teile der Aufgabe wie den Aufgabentext jedoch kann das LMS durchaus autark instanziieren. Besonders dann, wenn diese Aufgabenteile durch ein allgemein eingesetztes Aufgabenformat (wie ProFormA) in einer standardisierten Form vorliegen. Für Grader-spezifische, nicht standardisierte Bestandteile einer Aufgabe wird das LMS jedoch Backendsysteme in die Instanziierung einbeziehen. Die folgende Grafik stellt dar, wie das LMS bei Bedarf auf einen sog. Instanziierungsdienst zurückgreifen könnte. Der Instanziierungsdienst erzeugt aus einer Aufgabenschablone und einer vom LMS vorgegebenen Auflösung der Variationspunkte (composite variation resolution, kurz cvr) eine Aufgabeninstanz. Ggf. bedient sich der Instanziierungsdienst dabei weiterer Unterstützung des betroffenen Graders, der hierfür eine spezielle Instanziierungsfunktion anbietet. Es ist denkbar, dass eine Aufgabenschablone nicht als ganze, sondern dass nur einzelne Artefakte einer Aufgabenschablone zwischen den Systemen kommuniziert werden. Vorgelagerte Systeme wie das LMS oder der Instanziierungsdienst könnten dann Instanzen einzelner Artefakte selbständig zu einer ProFormA-Aufgabeninstanz verschmelzen.

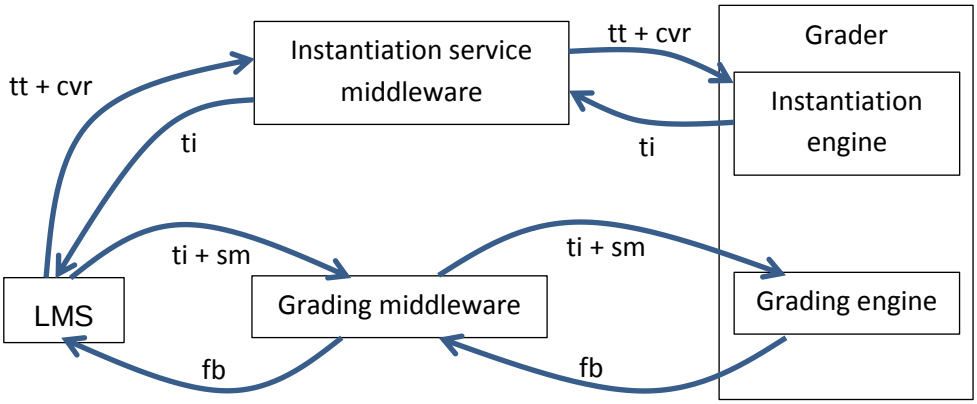


Abbildung 5: Nutzung eines Instanziierungsdienstes. Verwendete Abkürzungen: tt = task template (Aufgabenschablone), ti = task instance (Aufgabeninstanz), cvr = composite variation resolution (Auflösung aller Variationspunkte zu konkreten Werten), sm = submission (Einreichung), fb = Feedback.

3.5.2 Proprietäre Wertemengen eines Variationspunkts

In dem oben beschriebenen Szenario erzeugt das LMS eine Wertbelegung „cvr“ entweder zufallsbasiert oder mit Unterstützung eines der Lehrkraft präsentierten Dialoges. Sowohl die dialogbasierte als auch die zufallsbasierte Vergabe einer Wertebelegung greift auf die potentiell möglichen Werte je Variationspunkt zurück. Im Dialog dienen die möglichen Werte

der benutzerfreundlichen Auswahl, in der zufallsbasierten Vergabe wird aus allen möglichen Werten automatisch ein zufälliger Wert ausgewählt.

Es gibt Aufgabenschablonen, die nicht alle möglichen Wertebelegungen in standardisierter Form beinhalten. Die obige Beispielaufgabe gibt leider keine Hinweise darauf, dass dies möglich wäre. Wir illustrieren an zwei weiteren Aufgabentypen, wann die Wertemenge eines Variationspunkts abhängig von einer bestimmten Grader-Instanz sein wird. Erstens wäre eine Aufgabe denkbar, bei der eine Zeichenkodierung als Variationspunkt fungiert. Alle in der entsprechenden Java-Installation des Graders vorhandenen Zeichenkodierungen[5] stünden als potentielle Wertemenge dieses Variationspunktes zur Verfügung. Diese Wertemenge ist spezifisch für die Grader-Installation und dem LMS nicht ohne weiteres bekannt. Zweitens ist eine SQL-Aufgabe denkbar, die sich auf einen sehr großen Datenbestand in einer beim Grader installierten Datenbank stützt. Alle Einträge einer bestimmten Tabellenspalte könnten als Wertemenge eines Variationspunktes einer individualisierbaren Aufgabe dienen. Diese Wertemenge ist nur dem Grader bekannt, sie könnte von Graderinstanz zu Graderinstanz variieren und sie ist zudem zu groß, als dass es praktikabel wäre, diese vollständig in die Aufgabenschablone zu integrieren.

Für eine zufallsbasierte Belegung muss das LMS die zur Auswahl stehenden Werte je Variationspunkt kennen oder zumindest deren wie auch immer gearteten Identifier. Und auch die dialogbasierte Vergabe von Werten kann nur dann besonders benutzerfreundlich erfolgen, wenn das LMS alle möglichen Wertbelegungen der einzelnen Variationspunkte kennt und diese zur Auswahl anzeigt. Kennt das LMS nicht alle möglichen Wertbelegungen, ist immerhin eine manuelle Eingabe durch den Dozenten möglich. Diese ist allerdings erheblich fehleranfälliger und mühsamer.

Als erste Lösungsmöglichkeit könnte das LMS die unbekannte Wertemenge einfach beim Grader (via Instanziierungsdienst) erfragen. Ist die Wertemenge jedoch wie bei der SQL-Aufgabe sehr groß, wäre dies keine praktikable Option. Für die Liste der verfügbaren Zeichenkodierungen jedoch wäre dies ein gangbarer Weg (vgl. Abbildung 6).

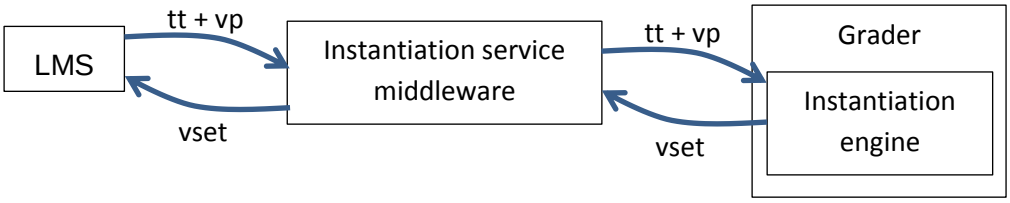


Abbildung 6: Abfrage der Variantenmenge (vset = variant set) für einen gegebenen Variationspunkt (vp) und eine gegebene Aufgabenschablone (tt)

Für eine sehr große Wertemenge, die nicht zum LMS übertragen werden kann und soll, muss die Wertbelegung dialogbasiert durch manuelle Eingabe erfolgen. Vertippt sich der Dozent, würde im Ablauf der Abbildung 5 der Instanziierungsdienst oder der Grader eine Falscheingabe feststellen und zurück weisen.

Zufallsbasiert kann das LMS im Falle sehr großer, dem LMS nicht bekannter Wertemengen die entsprechenden Variationspunkte einfach offen lassen. Abbildung 7 zeigt ein Szenario, in dem das LMS ein „pcvr“-Objekt an den Instanziierungsdienst übergibt. Diese „partial composite variation resolution“ enthält nicht alle notwendigen Wertbelegungen, sondern lediglich diejenigen, die das LMS selbständig zufallsbasiert vornehmen konnte. Auflösungen für die fehlenden Variationspunkte ergänzt nun der Grader selbständig. Die resultierende Gesamt-cvr erhält das LMS zusammen mit der Aufgabeninstanz zurück.

[5] Abfragbar z. B. durch `Charset.availableCharsets` (vgl. https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html#availableCharsets--)

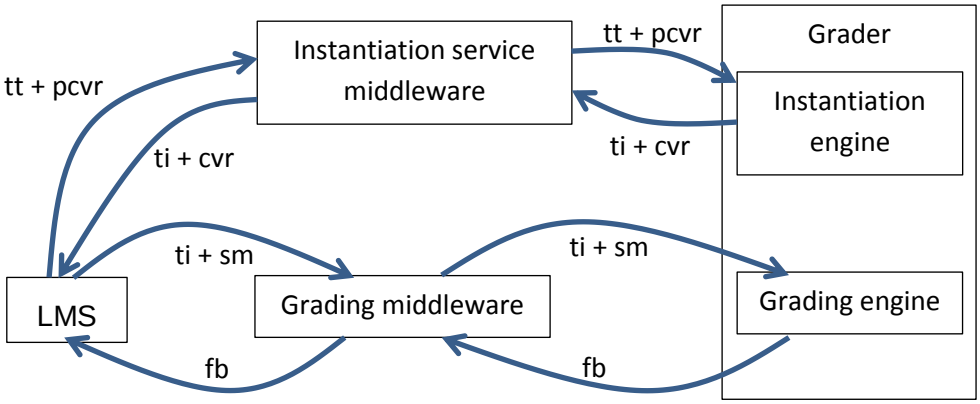


Abbildung 7: Abweichende Nutzung des Instanziierungsdienstes für eine zufallsbasierte Wahl einer Variante. Verwendete Abkürzung: pcvr = partial composite variation resolution.

Nachdem wir nun beschrieben haben, in welchem Systemumfeld Aufgabenschablonen und -instanzen eingesetzt werden könnten, sollen die weiteren Abschnitte beschreiben, wie Variationspunkte, deren Werte und ihre Abhängigkeiten beschrieben werden können.

4 Spezifikation von Variationspunkten und deren Abhängigkeiten

In diesem Abschnitt geben wir einen Überblick über die Art und Weise, wie wir in dem oben beschriebenen Umfeld Variationspunkte und deren Abhängigkeiten spezifizieren. Wir beziehen uns dabei auf die oben eingeführte Beispielaufgabe.

4.1 Benutzerschnittstelle

Einen Dialog, der einer Lehrkraft erlaubt, Variablenwerte festzulegen, zeigt Abbildung 8. Der Benutzer bedient die Regler beginnend mit dem obersten. Wenn zu einer in den oberen Zeilen getätigten Auswahl Abhängigkeiten zu weiter unten stehenden Variablen bestehen, passt der Dialog die in den unteren Zeilen auswählbaren Werte automatisch an. So kann sich der Benutzer Schritt für Schritt von oben nach unten durch alle Variablen „durcharbeiten“, ohne Gefahr zu laufen, eine ungültige Wertkombination auszuwählen. Variationspunkte, die keine Auswahl erlauben (bspw. „output“) werden ohne Eingabemöglichkeit einfach nur dargestellt.

Beispielsweise wählt der Benutzer für *domain* den Wert „numbers“. Automatisch verändert sich die Darstellung des Reglers für *className*, der nun drei Auswahloptionen anbietet (erkennbar an den Einrastpositionen). Nach Auswahl von „Figures“ schiebt der Benutzer als nächstes den Schieberegler für *set* nach rechts. Statt vorher drei sind nun zwei wählbare Optionen verfügbar („0-9“ und „0-9, I, V, X, L, C, D, M“).

Wir haben uns für einen Schieberegler als Interaktionselement entschieden, weil dieses auch dann kompakt und ressourcenschonend auf dem Bildschirm darstellbar ist, wenn der betreffende Variationspunkt eine sehr große Variantenmenge zur Auswahl stellt. Das Bedienelement muss zu Beginn lediglich die Anzahl der Werteoptionen kennen, um dann bei jedem Schiebeereignis den zur aktuellen Reglerposition zugehörigen Wert *on the fly* zu berechnen. Bedingung für die Eignung des Schiebereglers ist, dass sich die Werte jedes Variationspunkts durch einen fortlaufenden Index aufzählen lassen und dass der zu einem gegebenen Index gehörige Wert effizient ermittelbar ist. Die in den folgenden Abschnitten beschriebene Spezifikation der Wertemenge erfüllt diese Bedingung.

Derzeit ist der Dialog in JavaFX realisiert. Es ist geplant, diesen Dialog nach Javascript zu portieren, so dass eine Einbindung in webbasierte LMS einfach erfolgen kann. Für den Fall, dass dem LMS die Wertemenge eines Variationspunkts nicht bekannt ist (vgl. Abschnitt 3.5.2), müsste der Schieberegler zudem noch durch ein anderes Eingabefeld ersetzt werden (z. B. ein Textfeld).

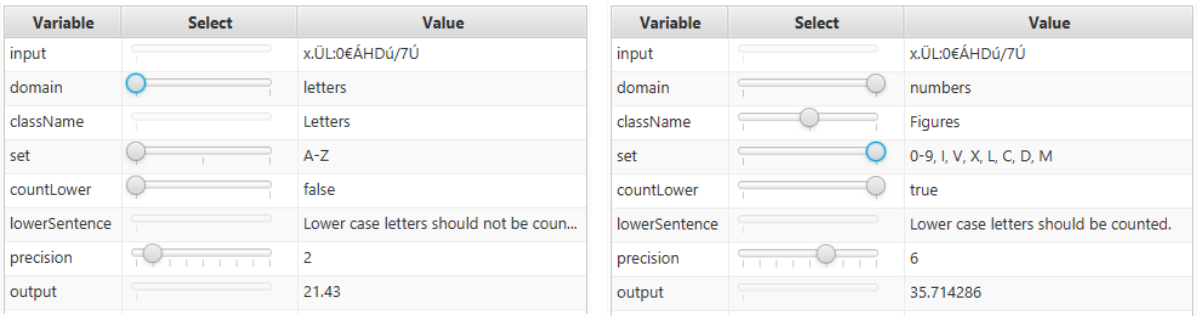


Abbildung 8: Dialog zur manuellen Auflösung aller Variationspunkte einer Aufgabe

4.2 Datenmodell

Ein Variationspunkt (Vp) besitzt einen Namen (key) und einen Datentyp (VpT). Eine Ausprägung des Vp ist eine Variante (V) mit einem Wert. Die entsprechenden Klassen Vp, VpT und V sind in Abbildung 9 dargestellt. Alle Variationspunkte einer Aufgabe zusammen sind in der Klasse CVp zusammengefasst. Werden alle Variationspunkte zwecks Erzeugung einer Aufgabeninstanz aufgelöst, entsteht ein CVr-Objekt. Das CVr-Objekt beinhaltet einen Vektor vom Typ CV, der je Variationspunkt die gewählte Variante speichert.

Eine Aufgabenschablone (tt = task template) enthält neben dem Aufgabentext (Abschnitt 3.2) und weiteren Artefakten (Abschnitt 3.4) eine Spezifikation der Variantenmenge. Jeder einzelne Variationspunkt wird hierin spezifiziert (VSpec). Die Komposition der Spezifikationen aller Variationspunkte der Aufgabe bezeichnen wir als CVSpec (composite variation specification). Ein solches CVSpec-Objekt deklariert alle Variationspunkte, die zugehörigen Wertemengen und deren Abhängigkeiten. Das CVSpec-Objekt fungiert als Wurzelknoten einer Hierarchie von CVSpecNode-Objekten (s. Abbildung 9). Es handelt sich bei dieser Hierarchie prinzipiell um eine Beschreibung einer Teilmenge des durch mehrere Variationspunkt-Dimensionen aufgespannten Raums. Die Teilmenge entsteht zunächst durch geschachtelte Vereinigungsmengen (Collect…) und kartesische Produkte (Combine…). Zur Vermeidung von Redundanz und zur Vermeidung langer Auflistungen (Anforderung (3)) sind spezielle Knotentypen wie Definitions- und Referenzierungsknoten (Def, Ref), Bereichsknoten (Range) und Ableitungsknoten (Derivation) ergänzt worden. Der Wurzelknoten besitzt mit dem Attribut „cvp“ eine Spezifikation aller Variationspunkte der Aufgabe. Nachfolgende Knoten speichern die für den jeweiligen Teilraum geltenden Variationspunkte[6].

[6] Das entsprechende Attribut ist optional, da sich jeder Knoten die zugehörigen Variationspunkte leicht vom Vaterknoten ermitteln lassen kann. Verpflichtend ist die Angabe der Variationspunkte dann, wenn lokale Umordnungen von Variationspunkten in Kindknoten gewünscht sind, um die Spezifikation kompakter zu gestalten. Mehr Details hierzu findet man in [18].

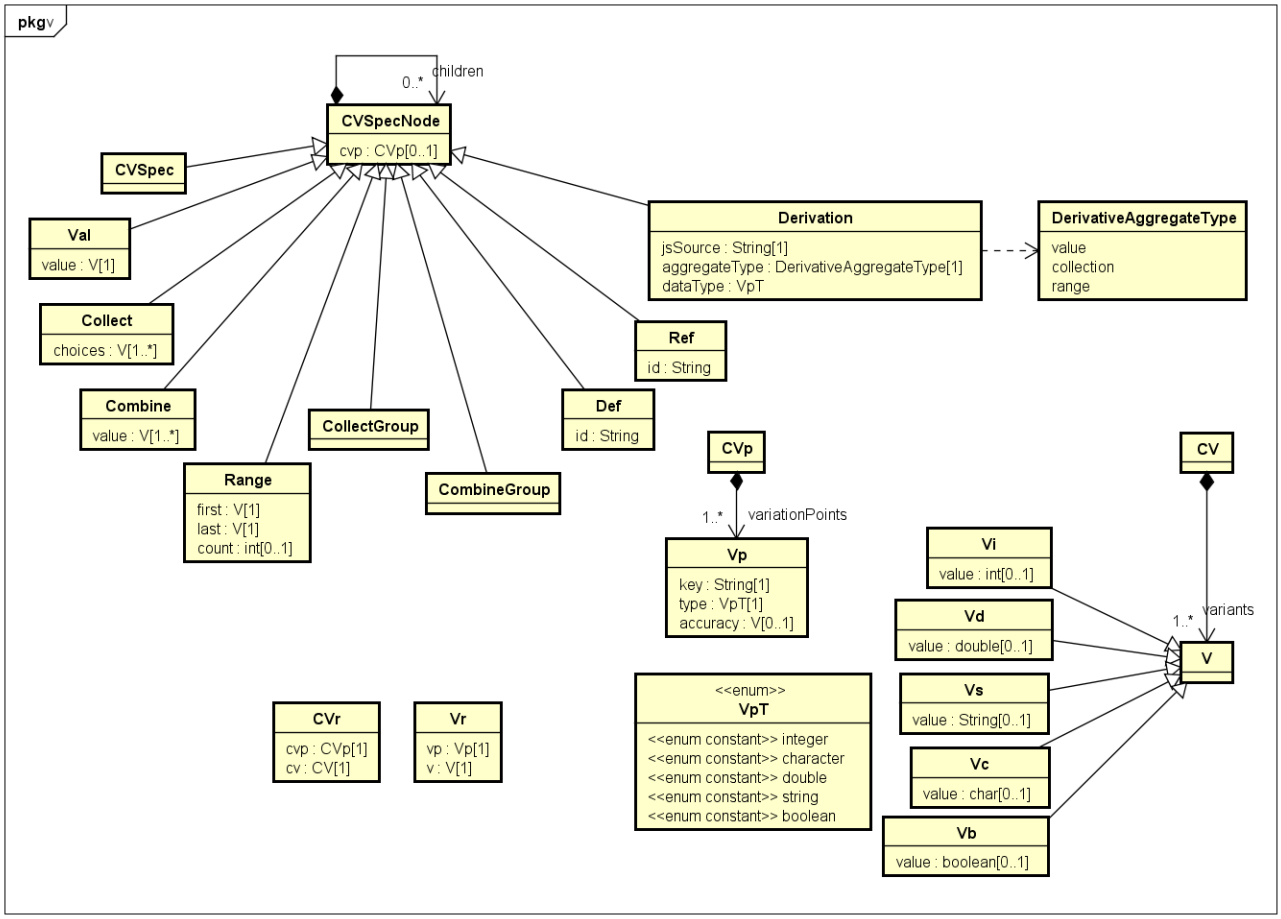


Abbildung 9: UML-Diagramm der Objekte zur Spezifikation von Variationspunkten und deren Abhängigkeiten. Die Menge unterstützter Datentypen (VpT) ist beliebig erweiterbar.

Die in Abbildung 9 dargestellten Datentypen von Variationspunkten (VpT) sind nur als kleine, erweiterbare Auswahl zu verstehen. Neben den Standarddatentypen (int, double, boolean, etc.) sind auch komplexere Datentypen denkbar. Diese lassen sich jedoch in der Regel auf einfache Datentypen zurückführen:

- Bezeichnet ein Variationspunkt eine Implementationsvariante einer durch die Studierenden zu nutzende Bibliotheksfunktion? Studierende sollen abhängig von der zugewiesenen Variante mit einer sich unterschiedlich verhaltenden, vorgegebenen Bibliotheksfunktion konfrontiert werden? Dann könnte der Variationspunkt vom Typ java.lang.reflect.Method sein, der sich leicht auf den Datentyp String durch Angabe des Klassen- und Methodennamens zurückführen lässt.
- Bezeichnet der Variationspunkt eine Inhaltsvariante einer durch das studentische Programm zu lesenden oder zu schreibenden Binärdatei? Dann könnte der Variationspunkt vom Array-Typ byte[] sein. Dieser lässt sich auf den Datentyp String zurückführen, indem eine Datei-ID angegeben wird, die auf eine oder mehrere mit der Aufgabenschablone verknüpfte Dateien verweist (ProFormA-Element „filref“).
- Bezeichnet der Variationspunkt die Schwierigkeit einer Aufgabe, die sich im konkreten Fall dadurch ausdrückt, dass bei der einfachen Aufgabenvariante lediglich zwei Methoden „a“ und „b“ realisiert werden müssen, in einer schwierigeren Variante jedoch zusätzlich die Methode „c“? Dann könnte der Variationstyp vom Array-Type String[] sein. Entweder erweitert man dazu die zur Verfügung stehenden VpT-Werte entsprechend um Array-Typen, oder man codiert das zu speichernde String-Array in einem String-Objekt.
- Sind abhängig vom Variationspunkt verschiedene Gewichtungen für einzelne Bewertungsaspekte einer Aufgabe zu vergeben? Beispielsweise indem in der

Anfängervariante Effizenzaspekte gegenüber Aspekten der korrekten Funktion untergewichtet werden? Dann könnte der Typ des Variationspunktes ein Array-Typ für Gewichte, z.B. double[] sein. Oder eine Map, die IDs der Bewertungsaspekte auf Bewertungsgewichte abbildet. Auch hier wäre eine Erweiterung der VpT-Ausprägungen denkbar oder eine Codierung in einem String.

Weitere Typen sind denkbar. In diesem Aufsatz werden wir uns ohne Beschränkung der Allgemeinheit der Ausführungen auf einfache Standarddatentypen beschränken.

4.3 Spezifikation der Beispielaufgabe in XML und Javascript

Das im vorangegangenen Abschnitt mit UML-Mitteln beschriebene Fachdatenmodell der gesuchten Spezifikation von Variationspunkten setzen wir mit den Sprachen XML und Javascript um. Beispielhaft geben wir unten in Quelltext 6 ein XML-Dokument an, das die Variationspunkte der Aufgabe 3 beschreibt. Unsere Beispielaufgabe ist geeignet, viele, aber nicht alle Möglichkeiten der vorgeschlagenen Notation zu illustrieren. Für eine vollständige Darstellung aller Möglichkeiten verweisen wir auf [18].

Das XML-Dokument in Quelltext 6 beginnt innerhalb des Wurzelelements mit der Angabe der Variationspunkte. Danach kommen die ineinander verschachtelten CombineGroup- und CollectGroup-Knoten des CVSpec-Baums. Blattknoten definieren zum einen konkrete Variationspunktwerte innerhalb eines <val>-Elements (z. B. „Letters“). Wenn mehrere Skalare zu einer Menge oder zu einem Tupel zusammengefasst werden sollen, verwenden wir statt <val> die Elemente <collect>, <combine> oder <range> (vgl. Tabelle 3). Ableitungsvorschriften werden als Javascript-Funktion in einem <derive>-Element angegeben. Zur Demonstration[7] wurde außerdem ein Def-Knoten (id1) für die Variable „countLower“ eingefügt, der von zwei Stellen aus referenziert wird.

	Kurzschreibweise	**Äquivalente Schreibweise**

Vereinigungsmenge mehrerer Skalare	`<collect>`
	`  <boolean value="true"/>`
	`  <boolean value="false"/>`
	`</collect>`
		`<collectGroup>`
		`  <val><boolean value="true"/></val>`
		`  <val><boolean value="false"/></val>`
		`</collectGroup>`

Verknüpfung mehrerer Skalare zu einem Tupel	`<combine>`
	`  <string value="0-9"/>`
	`  <boolean/>`
	`</combine>`
		`<combineGroup>`
		`  <val><string value="0-9"/></val>`
		`  <val><boolean/></val>`
		`</combineGroup>`

Vereinigungsmenge mehrerer äquidistanter Skalare	`<range>`
	`  <vRange>`
	`  <firstInteger value="1"/>`
	`  <lastInteger value="7"/>`
	`  <count>4</count>`
	`  </vRange>`
	`</range>`
		`<collectGroup>`
		`  <val><integer value="1"/></val>`
		`  <val><integer value="3"/></val>`
		`  <val><integer value="5"/></val>`
		`  <val><integer value="7"/></val>`
		`</collectGroup>`

Tabelle 3

[7] Ein Define-Knoten „lohnt“ sich an dieser Stelle in diesem kleinen Beispiel kaum und dient hier nur der Demonstration des Prinzips.

```
  :  <?xml version="1.0" ?>
  :  <v:cvSpec xmlns:v="urn:to-be-specified">
  :    <v:cvp>
  :      <v:vp key="input" type="string"></v:vp>
  :      <v:vp key="domain" type="string"></v:vp>
  :      <v:vp key="className" type="string"></v:vp>
  :      <v:vp key="set" type="string"></v:vp>
  :      <v:vp key="countLower" type="boolean"></v:vp>
  :      <v:vp key="lowerSentence" type="string"></v:vp>
  :      <v:vp key="precision" type="integer"></v:vp>
  :      <v:vp key="output" type="string"></v:vp>
  :    </v:cvp>
 1:    <v:combineGroup>
 2:      <v:val>
 3:        <v:string value="x.ÜL:0€ÁHDú/7Ú"></v:string>
  :      </v:val>
 4:      <v:collectGroup>
 5:        <v:define id="id1">
  :          <v:cvp>
  :            <v:vp key="countLower" type="boolean"></v:vp>
  :          </v:cvp>
 6:          <v:collect>
 7:            <v:boolean value="true"></v:boolean>
 8:            <v:boolean value="false"></v:boolean>
  :          </v:collect>
  :        </v:define>
 9:        <v:combineGroup>
10:          <v:val>
11:            <v:string value="letters"></v:string>
  :          </v:val>
12:          <v:val>
13:            <v:string value="Letters"></v:string>
  :          </v:val>
14:          <v:collect>
15:            <v:string value="A-Z"></v:string>
16:            <v:string value="A-Z, Ä, Ö, Ü"></v:string>
17:            <v:string value="A-Z, Œ"></v:string>
  :          </v:collect>
18:          <v:ref id="id1"></v:ref>
  :        </v:combineGroup>
19:        <v:combineGroup>
20:          <v:val>
21:            <v:string value="numbers"></v:string>
  :          </v:val>
22:          <v:collect>
23:            <v:string value="Numbers"></v:string>
24:            <v:string value="Figures"></v:string>
25:            <v:string value="Digits"></v:string>
  :          </v:collect>
26:          <v:collectGroup>
27:            <v:combine>
28:              <v:string value="0-9"></v:string>
29:              <v:boolean></v:boolean>
  :            </v:combine>
30:            <v:combineGroup>
31:              <v:val>
32:                <v:string value="0-9, I, V, X, L, C, D, M">
  :                                                    </v:string>
  :              </v:val>
33:              <v:ref id="id1"></v:ref>
  :            </v:combineGroup>
  :          </v:collectGroup>
  :        </v:combineGroup>
  :      </v:collectGroup>
34:      <v:derive dataType="string" aggregateType="value">
35:        <v:jsSource>/**
  :   * Calculates a new variation point value from other
  :   * variation point values.
  :   * @param {Object} obj - an object with variation point
  :   *        values
  :   * @param {String} obj.countLower - true, false or null
  :   *        indicating whether lower case letters should be
  :   *        counted
  :   * @returns {String} a sentence describing requirements
  :   *        regarding lower case letters
  :   */
  :  function apply(obj) {
  :    "use strict"
  :    if (obj.countLower === null)
  :      return "";
  :    if (obj.countLower === true)
  :      return "Lower case letters should be counted.";
  :    return "Lower case letters should not be counted.";
  :  };
  :  </v:jsSource>
  :      </v:derive>
36:      <v:range>
37:        <v:vRange>
  :          <v:firstInteger value="1"></v:firstInteger>
  :          <v:lastInteger value="9"></v:lastInteger>
  :          <v:steps>9</v:steps>
  :        </v:vRange>
  :      </v:range>
38:      <v:derive dataType="string" aggregateType="value">
39:        <v:jsSource><![CDATA[/**
  :   * Calculates a new variation point value from other
  :   * variation point values.
  :   * @param {Object} obj - an object with variation point
  :   *        values
  :   * @param {String} obj.input - input text
  :   * @param {String} obj.set - set of characters to be
  :   *        considered
  :   * @param {Boolean} obj.countLower - if true, then must
  :   *        count lower case letters
  :   * @param {Number} obj.precision - number of decimal places
  :   * @returns {String} percentage value with decimal digits as
  :   *          string
  :   */
  :  function apply(obj) {
  :    "use strict"
  :    var input= obj.input;
  :    if (obj.countLower === true) input= input.toUpperCase();
  :    var regex= new RegExp("["+obj.set.replace(/, /g, "")+"]");
  :    var cnt= 0;
  :    for (var i = 0, len = input.length; i < len; i++) {
  :        if (regex.test(input.charAt(i))) cnt++;
  :    }
  :    return (cnt * 100 / input.length).toFixed(obj.precision)
  :  };
  :  ]]></v:jsSource>
  :      </v:derive>
  :    </v:combineGroup>
  :  </v:cvSpec>
```

Quelltext 6: XML-Spezifikation der Variantenmenge für Aufgabe 3 (aus Platzgründen zweispaltig dargestellt). Die führenden Nummern werden wir in Abschnitt 4.5 aufgreifen.

4.4 Visualisierung kartesischer Produkte und Vereinigungsoperatoren

Zur Illustration besprechen wir den folgenden Teilausschnitt des vorstehenden XML-Dokuments:

```
  :  ...
 4:  <v:collectGroup>
  :    ...
 9:    <v:combineGroup>
10:      <v:val>
11:        <v:string value="letters"></v:string>
  :      </v:val>
12:      <v:val>
13:        <v:string value="Letters"></v:string>
  :      </v:val>
  :      ...
  :    </v:combineGroup>
19:    <v:combineGroup>
20:      <v:val>
21:        <v:string value="numbers"></v:string>
  :      </v:val>
22:      <v:collect>
23:        <v:string value="Numbers"></v:string>
24:        <v:string value="Figures"></v:string>
25:        <v:string value="Digits"></v:string>
  :      </v:collect>
  :      ...
  :    </v:combineGroup>
  :    ...
  :  </v:collectGroup>
  :  ...
```

Quelltext 7

Dieser Teilausschnitt betrifft lediglich die Variationspunkte *domain* und *className*. Die sinnvollen Wertkombinationen lassen sich als Teilmenge des zweidimensionalen Raums String $\times$ String auffassen. Die Variable *domain* besitzt die sinnvollen Werte {„letters“, „numbers“}, die Variable *className* die sinnvollen Werte {„Letters“, „Numbers“, „Figures“, „Digits“}. Die beiden Variationspunkte sind voneinander abhängig. Man kann sich die Teilmenge der sinnvollen Varianten als einen Bereich vorstellen, der durch „Slicing“ der Ebene in beiden Dimensionen entsteht. Abbildung 10 hebt die resultierende sinnvolle Variantenmenge grafisch durch dunkle kleine Quadrate hervor. Insgesamt resultieren in diesem Beispiel vier sinnvolle Varianten. Die sinnvollen Varianten lassen sich als Vereinigungsmenge zweier kartesischer Produkte wie folgt beschreiben

$$(\{\text{„numbers”}\} \times \{\text{„Numbers“, „Figures“, „Digits“}\}) \cup (\{\text{„letters”}\} \times \{\text{„Letters“}\}).$$

Im obigen XML-Fragment entspricht der *collectGroup*-Rahmen dem $\cup$-Operator und die geschachtelten *combine-Group*-Elemente den $\times$-Operatoren. Die weiter eingeschachtelten Elemente *val* und *collect* entsprechen den Mengenklammern { und }. Die *string*-Elemente entsprechen den in Anführungszeichen gesetzten Literalen.

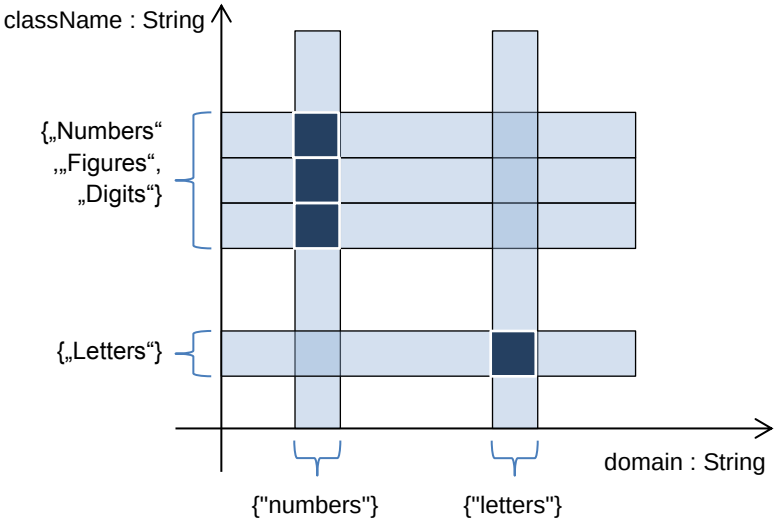


Abbildung 10: Visualisierung der sinnvollen Werte

4.5 Visualisierung als AND-XOR-Baum

Ein CVSpec-Objekt lässt sich zusammen mit seinen Nachfolgerknoten als Baum visualisieren. Die Visualisierung erfolgt lediglich zur Illustration. Daher wird die graphische Notation hier auch nicht präzise definiert. Die Visualisierung ist nicht Gegenstand irgendeines Prozesses der Prozesskette (vgl. Abschnitt 3.5).

Abbildung 11 zeigt einen Baum, der das in Abschnitt 4.3 abgedruckte XML-Dokument zu Aufgabe 3 visualisiert. Die rautenförmig gerahmten Nummern korrespondieren mit Zeilennummern in Quelltext 6. Viele innere Knoten des Baums sind vom Typ „AND“ (Trapezsymbol ⏢) und „XOR“ (umgekehrtes Trapezsymbol ⏣). Deshalb handelt es sich um einen AND-XOR-Baum. Die AND-Knoten korrespondieren mit „Combine“ und „CombineGroup“-Objekten und bezeichnen die Zusammensetzung eines Tupels aus einzelnen Variationspunkten. AND-Knoten repräsentieren also kartesische Produkte. Die XOR-Knoten benennen die Namen der jeweils betroffenen Variationspunkte und korrespondieren mit „Collect“, „CollectGroup“, „Val“, „Range“ und „Derivation“-Objekten. XOR-Knoten repräsentieren Vereinigungsoperationen der jeweiligen, von den Kindknoten repräsentieren Wertemengen für die im XOR-Knoten genannten Variablen.

Wir „lesen“ den Baum: Der Wurzel-AND-Knoten (Nr. 1) mit seinen 5 Kindknoten sagt aus, dass sich die gesamte Wertemenge aufzählen lässt, wenn man Wertemengen für die Variablen (*input)*, (*domain*, *className*, *set*, *countLower)*, (*lowerSentence)*, (*precision)* und (*output)* aufzählt und diese geeignet zu achtdimensionalen Tupeln zusammensetzt. Der Zusammensetzungsprozess muss alle Kindknoten einbeziehen, da es sich um einen AND-Knoten handelt. Es beginnt links mit einer einelementigen und eindimensionalen Menge S_2 für *input* (XOR-Knoten 2 vom Typ Val). Die Menge S_2 wird (da Knoten 1 ein AND-Knoten ist) kartesisch mit der vierdimensionalen Menge S_4 für (*domain*, *className*, *set*, *countLower*) multipliziert. Den Inhalt von S_4 betrachten wir im folgenden Absatz genauer. Zunächst bewegen wir uns zu den Geschwistern nach rechts weiter. Das Ergebnis des kartesischen Produkts S_2 x S_4 nennen wir $S_{2⏢4}$. Es schließt sich ein XOR-Knoten 34 an, der mit einem Derivation-Knoten[8] korrespondiert. Dieser Knoten repräsentiert eine Operation, die $S_{2⏢4}$ mit einem Javascript-Fragment (Knoten 35) für *lowerSentence* „kombiniert“. Die Vorschrift zur Bildung der resultierenden sechsdimensionalen Menge $S_{2⏢4⏢34}$ ist hier kein einfaches kartesisches Produkt, da die Menge des zweiten Operanden *lowerSentence* nicht vorab bekannt ist. Die Vorschrift zur Bildung von $S_{2⏢4⏢34}$ ist wie folgt: Für jedes Element e aus

[8] Ein mit dem Derive-Knoten verwandtes Konzept stellen sog. VSpec Derivations in der Common Variability Language (CVL) dar [12].

$S_{2\sqcap 4}$, berechne den zugehörigen Wert apply(e) durch Ausführung der Javascript-Funktion und füge (e,apply(e)) zu $S_{2\sqcap 4\sqcap 34}$ hinzu. Hierbei handelt es sich nicht mehr um ein klassisches kartesisches Produkt zweier Mengen. Die Operation kann aber weiterhin als Kombinationsoperation zweier Variantenmengen aufgefasst werden. Weiter geht es mit dem XOR-Knoten 36, der mit einem Range-Knoten korrespondiert. Die hierin definierte eindimensionale Menge S_{36}={1, 2, …., 9} für *precision* wird wieder kartesisch zu $S_{2\sqcap 4\sqcap 34}$ multipliziert. Das Ergebnis ist eine siebendimensionale Menge $S_{2\sqcap 4\sqcap 34\sqcap 36} = S_{2\sqcap 4\sqcap 34} \times S_{36}$. Die letzte Operation für Knoten 38 gleicht der für *lowerSentence*: die Javascript-Funktion zu *output* wird auf jedes Element in $S_{2\sqcap 4\sqcap 34\sqcap 36}$ angewendet und zu einem achtdimensionalen Tupel der Gesamtergebnismenge kombiniert.

Werfen wir nun einen Blick auf den CollectGroup-Knoten 4, der die Vereinigungsmenge der Mengen $S_9 \cup S_{19}$ darstellt. Doch zunächst ist da der erste parallelogrammförmige Kindknoten 5 (Def-Knoten), der einen Bezeichner id1 definiert[9]. Dem Bezeichner wird eine Wertemenge S_6 für die Variable *countLower* zugeordnet, die sich aus den beiden an den Blattknoten 7 und 8 notierten Werten { true, false } zusammensetzt. Der mit *countLower* bezeichnete XOR-Knoten 6 besagt, dass bei der Auflösung des Variationspunkts *countLower* zu einem konkreten booleschen Wert die beiden Kinder nicht gleichzeitig gewählt werden können, sondern nur genau eines von diesen. Die beiden verbleibenden Kindknoten 9 und 19 des Knotens 2 sind AND-Knoten ($\sqcap$). Unter diesen beiden Knoten werden separat voneinander alle Wertkombinationen für (*domain*, *className*, *set*, *countLower*) aufgelistet, die ein schlüssiges Gesamtbild der Aufgabe für je einen der Anwendungsbereiche „letters“ und „numbers“ bilden. Der Anwendungsbereich „letters“ befasst sich mit dem Zählen von Buchstaben. Eine beispielhafte Aufgabeninstanz dieses Anwendungsbereichs ist die obige Aufgabe 1. Dagegen ist Aufgabe 2 ein Beispiel einer Aufgabeninstanz des Anwendungsbereichs „numbers“, in dem es um das Zählen von Ziffern geht. Knoten 9 repräsentiert als CombineGroup-Knoten ein kartesisches Produkt $S_9 = S_{10} \times S_{12} \times S_{14} \times S_{18}$. Die Mengen S_{10}={„letters“} und S_{12}={„Letters“} werden durch Knoten vom Typ Val repräsentiert, da sie jeweils nur aus einem Element bestehen. Der XOR-Knoten 14 ist ein Collect-Knoten mit der Wertemenge S_{14}={„A-Z“, „A-Z, Ä, Ö, Ü“, „A-Z, Œ“}. Der Knoten 18 erhält seinen Wertebereich S_{18}={true, false} durch Verweis auf die Wertemenge des Knotens 6. Wenden wir uns nach rechts dem AND-Knoten 19 zu, so ist als einzige konzeptionelle Neuheit dessen Nachfolger-Knoten 27 zu nennen, dessen XML-Pendant als <combine>-Element zwei Skalare zu der einelementigen und zweidimensionalen Wertemenge S_{27}={ („0-9“, null) } verknüpft.

[9] Mit dem Define-Knoten verwandt ist das Konzept von feature model references [19].

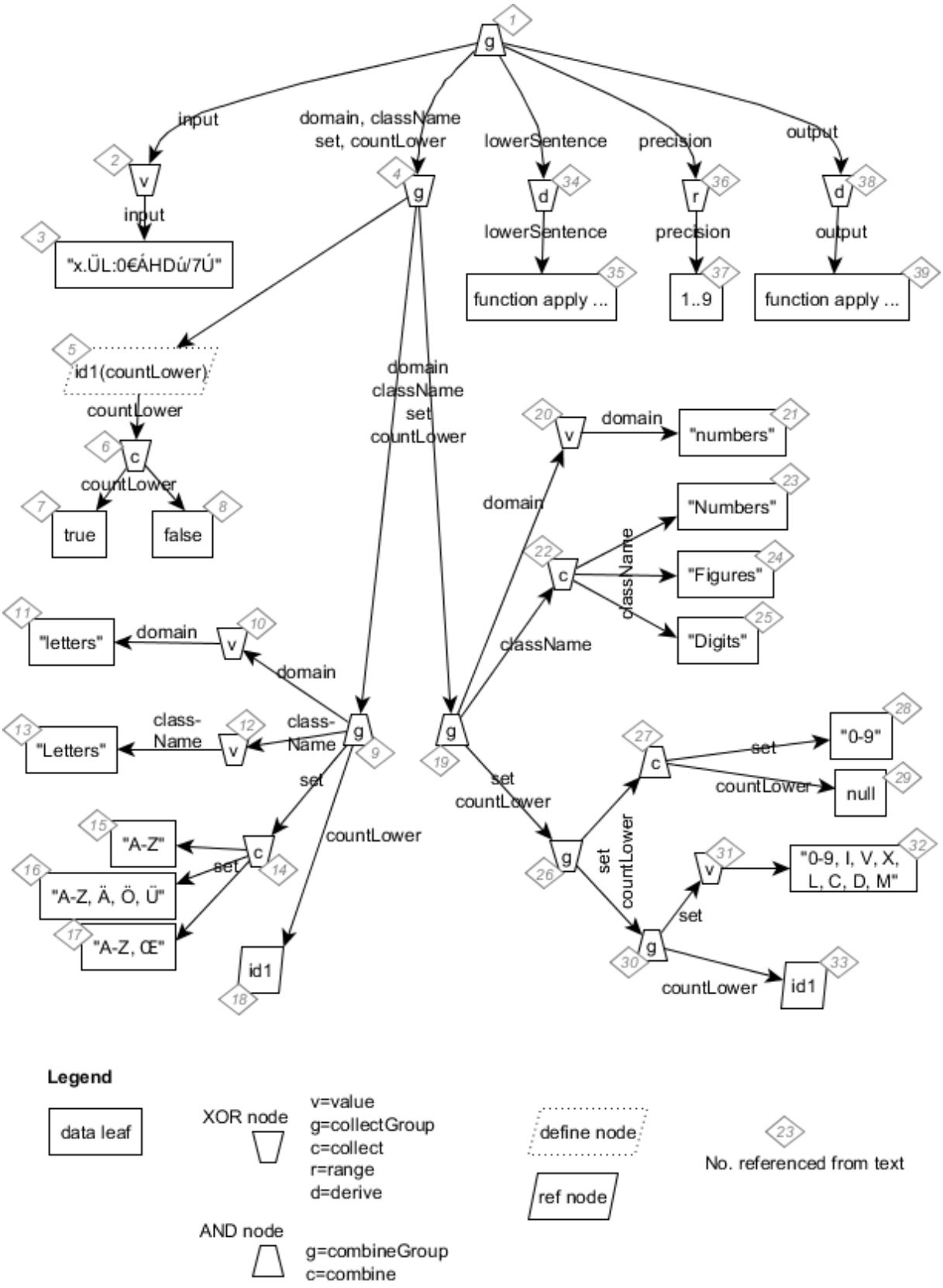


Abbildung 11: AND-XOR-Baum zu Quelltext 6.

Das hier besprochene Beispiel bietet leider keine Gelegenheit, Derivation-Knoten zu besprechen, die nicht nur einen einfachen Wert als Ergebnis der apply-Funktion liefern, sondern gleiche eine ganze Wertemenge. Abbildung 9 definiert als *DerivativeAggregateType* neben *value* weitere Ausprägungen. Die Ausprägung *range* wird noch nicht unterstützt. Die

Ausprägung *collection* jedoch wird unterstützt und erwartet, dass das entsprechende Javascript-Fragment eine Funktion apply enthält, die als Ergebnis ein Javascript-Array liefert. Die zusammengesetzte Wertemenge Z berechnet sich dann nach der folgenden Vorschrift: Für jedes Element x aus der Menge X, die durch die dem Derivation-Knoten vorangehenden Geschwisterknoten repräsentiert wird: berechne das zugehörige Array Y_x=apply(x) und füge dann alle Elemente { (x,y) : y aus Y_x } zu Z hinzu.

4.6 Java-Builder

Im Rahmen dieses Aufsatzes ist eine Java-Bibliothek entstanden, die die Erzeugung des in Quelltext 6 abgedruckten XML-Dokuments erleichtern kann. Im vorliegenden Beispiel der Aufgabe 3 sieht das wie unten abgedruckt aus. Der Java-Quelltext spiegelt genau die Struktur des AND-XOR-Baumes wider:

```
CVSpec.build(Vp.s("input"), Vp.s("domain"), Vp.s("className"), Vp.s("set"),
             Vp.b("countLower"), Vp.s("lowerSentence"), Vp.i("precision"), Vp.s("output"))
    .combineGroup()
        .val(Helper.DEFAULT_USER_INPUT) // input
        .collectGroup()
            .define("id1", "countLower")
                .collect(true, false)  // countLower
            .endDefine()
            .combineGroup()
                .val("letters") // domain
                .val("Letters") // className
                .collect("A-Z", "A-Z, Ä, Ö, Ü", "A-Z, Œ") // set
                .ref("id1")  // countLower
            .endCombineGroup()
            .combineGroup()
                .val("numbers") // domain
                .collect("Numbers", "Figures", "Digits") // className
                .collectGroup()
                    .combine("0-9", new VpBoolean(null)) // lower case letters requirement is not
                                                         // meaningful with set 0-9
                    .combineGroup()
                        .val("0-9, I, V, X, L, C, D, M")  // set
                        .ref("id1")  // countLower
                    .endCombineGroup()
                .endCollectGroup()
            .endCombineGroup()
        .endCollectGroup()
        .deriveVal(VpType.STRING, Helper.jsSrc_lowerSentence) // lowerSentence
        .range(1,9) // precision
        .deriveVal(VpType.STRING, Helper.jsSrc_output) // output
    .endCombineGroup()
.endBuild();
```

Quelltext 8

In diesem Java-Fragment wird auf Javascript-Quelltexte verwiesen (`jsSrc_lowerSentence` und `jsSrc_output`), die als Strings vorab mit genau dem Inhalt befüllt wurden, wie er bereits im XML-Dokument in Quelltext 6 ersichtlich ist.

Das Ergebnis des vorstehenden Java-Statements ist ein Java-Objekt vom Typ CVSpec (vgl. Abbildung 9). Die entsprechende Java-Klasse bietet JAXB-basierte Möglichkeiten an, das Objekt in einen XML-Strom zu serialisieren. Mit dieser Java-Bibliothek ist es einem Aufgabenautor nun auf komfortable Weise möglich, die Variationspunkte und ihre Wertemengen kompakt zu beschreiben.

Häufig ist es notwendig, dieselbe Funktionalität sowohl für die Definition der Wertemengen als auch für die JUnit-Tests zu implementieren. In unserem Beispiel ist das ersichtlich. Einerseits ist in Quelltext 6 im letzten `<v:jsSource>`-Element Javascript-Code zur Berechnung der Ausgabe enthalten. Andererseits wurde in Abschnitt 3.4 eine Java-Methode `Helper.getPercentage` mit derselben Funktionalität implementiert. Die Java-Methode lässt sich nun leicht durch einen Aufruf der Javascript-Funktion ersetzen. Graja setzt hierbei auf das inzwischen zur Java-Standardausstattung gehörende Nashorn-Framework zur Ausführung von Javascript-Code.

4.7 Effizienzbetrachtung

Die durch den Java-Builder (Abschnitt 4.6) erzeugte Datenstruktur, die wir in Abschnitt 4.5 als Baum visualisiert und in Abschnitt 4.3 als XML-Dokument abgedruckt haben, kann genutzt werden, um darauf aufbauend effiziente Datenstrukturen für verschiedene Anfragen zu erstellen. Insbesondere in dem in Abschnitt 4.1 beschriebenen Dialog müssen Index-Anfragen schnell beantwortet werden, um eine antwortfreudige Benutzeroberfläche zu erhalten. Zudem benötigen wir Zufalls-Anfragen für die Generierung einer zufällig gewählten neuen Variante. Wir skizzieren Algorithmen für mehrere Aufgaben. Wir gehen dabei davon aus, dass der CVSpec-Baum wesentlich weniger Knoten besitzt als die Menge der von diesem Baum repräsentierten Varianten. Vor allem durch Bereichsknoten (Range) kann die durch einen einzigen Knoten repräsentierte Wertemenge sehr groß werden. Für Suchanfragen im CVSpec-Baum sind effiziente Datenstrukturen daher weniger wichtig als für Suchanfragen in der Menge der repräsentierten Werte.

4.7.1 Indexanfrage für einen Variationspunkt

Nach einem Schieberegler-Ereignis (vgl. Abschnitt 4.1) für den Variationspunkt i sind zwei Operationen durchzuführen. Zum ersten ist im Variationspunkt i der zur neuen Reglerposition gehörende Wert zu ermitteln. Zum zweiten muss für den nachfolgende Variationspunkt i+1 die Wertemenge M_{i+1} ermittelt werden.

Wir speichern für jeden Variationspunkt k die zugehörige Menge auswählbarer Werte M_k. Die Wertemenge M_k kann insbesondere bei Verwendung von Wertebereichen (Range) sehr groß sein, so dass diese nicht vollständig aufgezählt wird, sondern in einer geeigneten effizienten Datenstruktur D_k dargestellt wird. D_k ist geeignet, wenn sie Indexanfragen bei Schieberegler-Ereignissen des Variationspunkts k effizient beantworten kann. Bei intervallskalierten Variationspunkten k nutzen wir für D_k in unserer Implementierung eine Variante eines Intervallbaums, in den wir Einzelwerte und Bereichsangaben (Range) einfügen. Für ordinalskalierte Variationspunkte k bietet sich als D_k eine herkömmliche, balancierte Suchbaumstruktur an. Diese Datenstrukturen sind geeignet, bei Hinzufügungen Wertduplikate effizient zu vermeiden und Indexanfragen effizient zu beantworten.

Nach einem abgeschlossenen Schieberegler-Ereignis für den Variationspunkt i bezeichne $T_{1:i}$ das Tupel der in den Variationspunkten 1 bis i gewählten Werte. Nun muss für den nachfolgenden Variationspunkt i+1 die Wertemenge M_{i+1} ermittelt werden. Für einen gegeben festen „Prefix“ $T_{1:i}$ suchen wir also die Menge der möglichen Werte in Variationspunkt i+1. Dazu führen wir einen Tiefendurchlauf durch den CVSpec-Baum durch. Währenddessen bauen wir sukzessive die die Menge M_{i+1} repräsentierende Datenstruktur D_{i+1} auf. Es muss beim Baumdurchlauf durch den CVSpec-Baum jeweils geprüft werden, ob der aktuell besuchte Knoten eine Wertemenge repräsentiert, die, wenn sie auf die Variationspunkte {1, … , i} projiziert wird, das Tupel $T_{1:i}$ enthält. Es ist daher eine „ist-enthalten“-Operation zu realisieren (s. u.). Sodann ist beim Baumdurchlauf an AND-Knoten jeweils das Kind auszuwählen, das den Variationspunkt i+1 repräsentiert. An XOR-Knoten sind alle Kindknoten abzuarbeiten. Projiziert man die Werte aller auf diese Weise besuchten Blattknoten auf den Variationspunkt i+1, ergeben sich Skalare, die sukzessive zu D_{i+1} hinzugefügt werden.

4.7.2 Ist-enthalten-Anfrage

Für ein gegebenes Tupel $T_{1:i}$ soll geprüft werden, ob dieses in der vom CVSpec-Baum repräsentierten und auf die Variationspunkte {1, …, i} projizierte Wertemenge enthalten ist.

Eine solche Operation wird einerseits im vorstehenden Abschnitt genutzt. Andererseits dient diese Operation der Prüfung der Gültigkeit einer manuell eingegebenen Wertebelegung (vgl. Abschnitt 3.5.2).

Dazu führen wir einen Tiefendurchlauf durch den CVSpec-Baum durch. An AND-Knoten wird das Tupel $T_{1:i}$ auf die jeweiligen von den Kindknoten repräsentierten Variationspunkte

projiziert und nach unten gereicht. An XOR-Knoten werden alle Kindknoten abgearbeitet. An Blattknoten kann dann ein Skalarvergleich durchgeführt werden.

4.7.3 Zufällige Wertebelegung

Es soll eine zufällige Wertebelegung aller Variationspunkte ermittelt werden. Dazu führen wir einen Tiefendurchlauf durch den CVSpec-Baum durch. An AND-Knoten muss jeder Kindknoten besucht werden, an XOR-Knoten jedoch nur ein zufällig gewählter Kindknoten.

Zu bemerken ist hier, dass eine gleichverteilte Wertebelegung nicht immer effizient möglich ist. Für Ableitungs-Knoten (Derivation) können wir vorab nicht wissen, wie viele verschiedene Skalare von dem Knoten repräsentiert werden. Daher ist es auch nicht immer möglich, durch geeignete Gewichtung der Kinder eines CollectGroup-Knotens für eine Verteilung zu sorgen, die jede mögliche Belegung mit gleicher Wahrscheinlichkeit auswählt. Denkbar und sinnvoll wäre, bei jedem Derivation-Knoten zusätzlich einen Korridor zu speichern (min/max), in dem sich die Größe der von der zugehörigen Javascript-Funktion berechneten Wertemenge bewegt. Eine solche Erweiterung des Datenmodells ist für die Zukunft geplant.

5 Zusammenfassung

In diesem Aufsatz haben wir eine Notation und zugehörige Implementierung gezeigt, wie Variationspunkte und deren Wertebereiche und Randbedingungen für individualisierbare Programmieraufgaben beschrieben werden können. Wir betrachteten die Nutzung von Variationspunkten im Aufgabentext einer Beispielaufgabe und den zugehörigen weiteren Aufgabenartefakten und diskutierten, wie die als Schablone vorliegende Aufgabe von den beteiligten Systemen instanziiert und bewertet werden kann. Parallelen zu den in der Produktlinienentwicklung üblichen Featurediagrammen wurden beispielhaft aufgezeigt.

Die vorgeschlagene Notation kann als Einteilung des durch n Variationspunkte aufgespannten n-dimensionalen Parameterraumes visualisiert werden. Der Raum wird durch Auswahl konkreter Werte in Scheiben geschnitten. Voneinander unabhängige Variationspunkte führen zum kartesischen Produkt von Teilmengen. Voneinander abhängige Variationspunkte führen zu einer Vereinigungsmenge mehrere kartesischer Produkte, wobei die einzelnen kartesischen Produkte unterschiedliche Variantenmengen der beteiligten Variationspunkte auswählen. Sind die Abhängigkeiten der beteiligten Variationspunkte nicht symmetrisch, kann durch Definition von Teilmengen und ihre Referenzierung eine redundanzarme Definition erreicht werden. Kann bei einem Variationspunkt p, der von einem oder mehreren anderen Variationspunkten q abhängt, leicht eine Berechnungsvorschrift angegeben werden, die die gültigen p-Teilmengen abhängig von den q-Werten ermittelt, so kann diese Berechnungsvorschrift in Gestalt einer Javascript-Funktion implementiert und direkt an geeigneter Stelle in das Gesamtgefüge aller Parameter eingehängt werden. Die Berechnungsvorschrift ist immer dann vorzuziehen, wenn diese einfacher formuliert werden kann, als die gültigen p/q-Tupel aufzulisten.

Die Notation ist in XML/Javascript realisiert und kann als AND-XOR-Baum visualisiert werden. Eine Java-Bibliothek zur bequemen Generierung einer XML-Datei wurde vorgestellt. Es wurde ein in JavaFX realisierter Benutzerdialog zur manuellen Auswahl einer Wertebelegung aller Variationspunkte gezeigt und dessen Anforderungen an effiziente Datenstrukturen analysiert. Die zufällige Auswahl einer Wertebelegung wird von dem Vorschlag unterstützt. Eine gleichverteilte zufällige Auswahl ist bei Verwendung von Javascript-Berechnungsvorschriften ggf. durch zusätzliche Maßnahmen effizient erreichbar.

Zukünftige Arbeiten sollten sich mit der Ausweitung des Vorschlags auf weitere Aufgaben und weitere Grader beschäftigen. Erweiterungen hinsichtlich Grader-spezifischer Wertemengen in einem Variationspunkt sollten in den Vorschlag integriert werden. Neben konzeptionellen Arbeiten sind die betroffenen Systeme (LMS, Middleware, Grader) systemtechnisch anzupassen und zu erweitern. Zudem sollte angestrebt werden, das

ProFormA-Aufgabenformat um einen Standard für die Spezifikation von Variationspunkten in variablen Programmieraufgaben zu erweitern.

Literaturverzeichnis

[1] Prechelt, L., Malpohl, G., Philippsen, M. (2002), Finding plagiarisms among a set of programs with JPlag, J. UCS, 8. Jg., Nr. 11, S. 1016.

[2] Sheard, J., Dick, M. (2003), Influences on cheating practice of graduate students in IT courses: what are the factors?, ACM SIGCSE Bulletin. Vol. 35. No. 3. ACM.

[3] Kashy, D. A., Albertelli, G., Ashkenazy, G., Kashy, E., Ng, H.-K., Theonnessen, M. (2001), Individualized interactive exercises: A promising role for network technology, in: Proceedings of the 31st ASEE/IEEE Frontiers in Education Conference (Reno, NV).

[4] Brusilovsky, P., Sosnovsky, S. (2005), Individualized exercises for self-assessment of programming knowledge: An evaluation of quizpack, ACM Journal of Educational Resources in Computing, Vol. 5, No. 3.

[5] Hsiao, I.-H., Brusilovsky, P., Sosnovsky, S. (2008), Web-based parameterized questions for object-oriented programming, in: Proceedings of World Conference on E-Learning, E-Learn.

[6] Radosevic, D., Orehavocki, T., Stapic, Z. (2010), Automatic On-line Generation of Student's Exercises in Teaching Programming, in: Central European Conference on Information and Intelligent Systems, Varazdin, S. 87-93.

[7] Spinellis, D., Zaharias, P., Vrechopoulos, A. (2007), Coping with plagiarism and grading load: Randomized programming assignments and reflective grading, Computer applications in engineering education 15.2, S. 113-123.

[8] Krishna, A. K., Kumar, A. N. (2001), A problem generator to learn expression: evaluation in CSI, and its effectiveness, Journal of Computing Sciences in Colleges 16.4, S. 34-43.

[9] Holohan, E., et al. (2006), The generation of e-learning exercise problems from subject ontologies, in: Sixth International Conference on Advanced Learning Technologies, IEEE.

[10] Alsubait, T., Parsia, B., Sattler, U. (2014), Generating Multiple Choice Questions From Ontologies: Lessons Learnt. OWLED.

[11] Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S. (1990), Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21.

[12] Haugen, Ø. (2012), Common Variability Language (CVL) – OMG Revised Submission. OMG document ad/2012-08-05.

[13] Garmann, R. (2017), Der Grader Graja, in: Bott, O. J., Fricke, P., Priss, U., Striewe, M. (Hrsg): Automatisierte Bewertung in der Programmierausbildung. Waxmann. Münster.

[14] Strickroth, S., Striewe, M., Müller, O., Priss, U., Becker, S., Rod, O., Garmann, R., Bott, O. J., Pinkwart, N. (2015), ProFormA: An XML-based exchange format for programming tasks, in: eleed, Nr. 11.

[15] Pohl, K., Böckle, G., van der Linden, F. (2005), Software Product Line Engineering: Foundations, Principles, and Techniques, Springer.

[16] Laguna, M. A., Marqués, J.M. (2009), Feature diagrams and their transformations: an extensible meta-model, in: Software Engineering and Advanced Application. SEAA'09. 35th Euromicro Conference on. IEEE.

[17] Garmann, R., Heine, F., Werner, P. (2015), Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme, in: DeLFI 2015.

[18] Garmann, R. (2018), Eine Java-Bibliothek zur Spezifikation von Variabilität in automatisch bewerteten Programmieraufgaben. Bericht, Hochschule Hannover. http://nbn-resolving.de/urn:nbn:de:bsz:960-opus4-11874

[19] Czarnecki, K., Kim, C.H.P. (2005), Cardinality-based feature modeling and constraints: a progress report, Proceedings of the International Workshop on Software Factories at OOPSLA, San Diego, California, USA.

[20] Svahnberg, M., Van Gurp, J.; Bosch, J. (2005), A taxonomy of variability realization techniques. Software: Practice and experience, 35(8), S. 705-754.